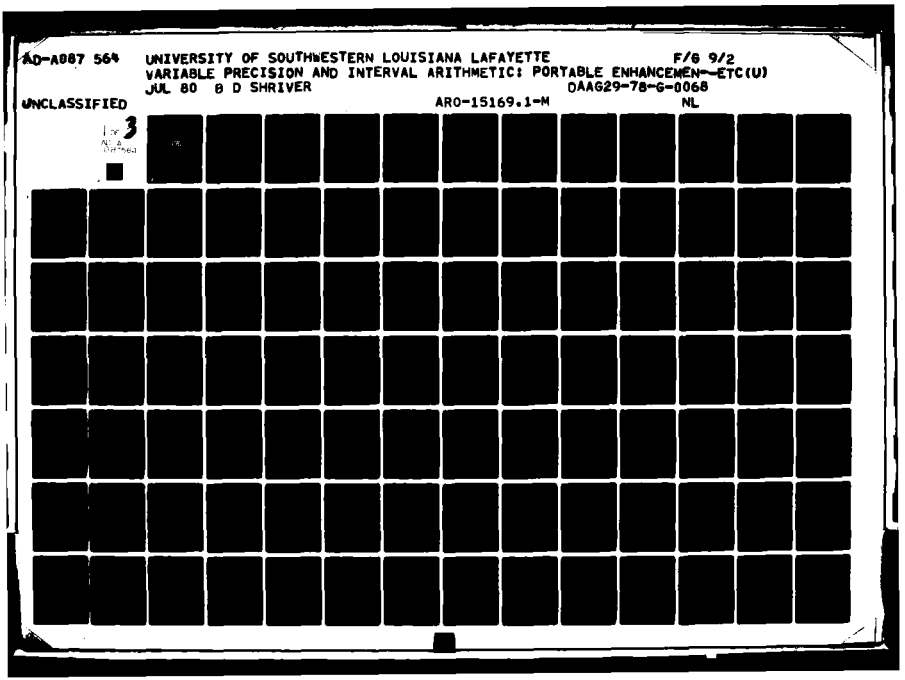
AD-A087 564 UNIVERSITY OF SOUTHWESTERN LOUISIANA LAFAYETTE F/G 9/2
VARIABLE PRECISION AND INTERVAL ARITHMETIC: PORTABLE ENHANCEMEN--ETC(U)
JUL 80 B D SHRIVER DAAG29-78-G-0068
UNCLASSIFIED ARO-15169.1-M NL



UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: 15169.1-M (ARO)
2. GOVT ACCESSION NO.: AD-A087564
3. RECIPIENT'S CATALOG NUMBER:
4. TITLE (and Subtitle): VARIABLE PRECISION AND INTERVAL ARITHMETIC: PORTABLE ENHANCEMENTS TO FORTRAN
5. TYPE OF REPORT & PERIOD COVERED: Final Report: 15 Mar 78 - 14 Mar 80
6. PERFORMING ORG. REPORT NUMBER:
7. AUTHOR(s): Bruce D. Shriver
8. CONTRACT OR GRANT NUMBER(s): DAAG29-78-G-0068
9. PERFORMING ORGANIZATION NAME AND ADDRESS: University of Southwestern Louisiana, Lafayetta, LA 70504
10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 2.70
11. CONTROLLING OFFICE NAME AND ADDRESS: U. S. Army Research Office, Post Office Box 12211, Research Triangle Park, NC 27709
12. REPORT DATE: Jul 80
13. NUMBER OF PAGES: 19+
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)
15. SECURITY CLASS. (of this report): Unclassified
15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

LEVEL II

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

NA

18. SUPPLEMENTARY NOTES

The view, opinions, and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

FORTRAN
programming languages
interval arithmetic
variable precision

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

It was the intent of the research reported here to develop, using preprocessing techniques, extensions to FORTRAN which would include: (1) the extended data types VPINTEGER, VPREAL, VPCOMPLEX and VPINTERVAL. The data types are analogous to those of standard FORTRAN and the single precision interval data type, but can be of arbitrary precision, (2) The standard built-in functions (SIN, COS, SQRT, etc.) for use with the new data types, and (3) The input/output facilities of standard FORTRAN extended for use with the extended data types.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED 407075

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ADA087564

DDC FILE COPY

80 8 1 063

FINAL REPORT

Variable Precision and Interval Arithmetic: Portable Enhancements to FORTRAN

Research Agreement No. DAAG29-78-G-0068

1. Background

The relationship between WES, USL and ARO began when WES wanted to examine interval arithmetic as a tool. It was hoped that interval arithmetic would serve as a valuable tool in ascertaining the reliability of values produced by application programs in use at WES. A FORTRAN implementation of interval arithmetic had been developed at MRC using the preprocessor AUGMENT [1,2,3,4,5]. This package was to be installed at both USL and WES for the purpose of determining whether or not interval arithmetic would be a tool consistent with the goals and interests of WES. The role that USL played concerned the implementing and benchmarking of the interval arithmetic package. This involved the conversion of application programs in use at WES so that real and double precision computations would be performed using interval arithmetic. The converted programs were then executed over a range of input values. The purpose was to gauge the value of interval arithmetic in the certification of each program's reliability. This evaluation proved successful in that a data sensitivity was uncovered in one of the programs. A

USL technical report [6] was written and published giving a description of this work.

During this period there were two separate interval arithmetic packages implemented at USL. The first of these was the MRC's single precision real interval package; the second was a 56 decimal digit interval package. The single precision real version of the interval arithmetic package represented values as two single precision real numbers. These two single precision real numbers are considered as bounds on an "exact" but possibly not machine representable value. This package is well described in [7].

The other interval arithmetic package, the 56 decimal digit interval version, was developed here at USL. This version was based on the 59 decimal digit arithmetic unit of the Honeywell H68/80 computer and represents the endpoints as two 56 decimal digit values. All testing of algorithms done under the single precision real version was also done under this version. The results and a description of the implementation are to be found in [8].

The fact that the 56 decimal digit interval arithmetic package was hardware based, as well as being written in PL/1, precluded its being transported to any other system. However, the results of work with this package provided valuable insights into the effects of extending the precision in which computations are

performed. It gave direction to the research that was to follow.

2. Goals and Accomplishments

It was the intent of the current work to develop, using preprocessing techniques, extensions to FORTRAN which would include:

1) the extended data types VPINTEGER, VPREAL, VPCOMPLEX and VPINTERVAL. The data types are analogous to those of standard FORTRAN and the single precision interval data type, but can be of arbitrary precision.

2) The standard built-in functions (SIN, COS, SQRT, etc.) for use with the new data types.

3) the input/output facilities of standard FORTRAN extended for use with the extended data types.

A users guide, which would detail experiences and suggestions on when and how to use variable precision arithmetic, was also produced.

The work was also to have accomplished the production of a comprehensive set of specifications for the organization of an

arithmetic unit which can efficiently support variable precision arithmetic.

As outlined in the grant proposal, FORTRAN was to have been extended using the preprocessor AUGMENT. Two extended FORTRAN languages were to have been constructed:

1) Variable Precision FORTRAN (VPFOR)

2) Variable Precision Interval FORTRAN (VPINTFOR)

VPFOR allows the use of VPREAL, VPINTEGER and VPCOMPLEX. VPINTFOR would allow the use of VPINTERVAL. Both languages would translate a program written in FORTRAN with the above data types using standard FORTRAN built-in functions and the standard FORTRAN input/output facilities of READ and WRITE into a standard FORTRAN program with imbedded subroutine calls which would execute the appropriate extended precision operations. These languages were to be "constructed" as "virtual compilers" that would translate the "virtual" source language, VPFOR for example, to the "virtual" object code, standard FORTRAN.

As with many research projects, the finished products do not quite conform to the specifications. There were those items which proved to involve more effort to implement than their relative importance justified, as well as those items whose inclusion greatly enhanced the overall product. The input/output extensions were not done as specified. Rather than allowing the specifications for input/output of extended data types to be

contained in FORTRAN FORMAT statements, the input/output was left as explicit calls but was simplified to be more usable. The RATFOR preprocessor [9] was attached to allow "structured programming" techniques by the user. Also the ease of specification of the precision of variables of the extended data types was accommodated to a higher degree than was envisioned in the proposal. The finished product also allows mixed mode arithmetic of all of the extended data types and their standard FORTRAN counterparts.

There were also changes in product definition dictated by the actual implementation. Since the base representation of all of the extended data types was very similar and, in fact, the same operations involving two different data types would call the the same subroutines at lower levels, only one FORTRAN language was constructed. This language, SEPAFOR (Structured Extended Precision Arithmetic FORtran), includes all of the extended data types. The next section details the history of the implementation of the variable precision data type.

3. Design and Implementation of SEPAFOR

The first major decision to be made in the design phase of the variable precision interval arithmetic package was how should the basic operations be implemented. There was already a multiple precision real package in use on Multics, and, in fact, had been

used by the 56 decimal digit interval package in implementing the standard built-in FORTRAN functions. A closer study of the package, developed by R. P. Brent of the Australian National University [10], indicated that the package was appropriate for use in the implementation of SEPAFOR. Its assets included:

1) Portability -- due to the use of integer arithmetic and standard FORTRAN the package was machine independent with the exception of two conversion subroutines.
2) Reliability -- the package had been thoroughly tested and had been in use for quite some time on several different machines.
3) Well constructed -- the package had been constructed with contemporary "structured programming" techniques. As a result it was highly modular with fairly good internal documentation which lent itself well to modification.
4) Availability -- it was already resident on the Multics system.

Once the decision was made to base the implementation on Brent's multiple precision package, the next problem to arise was in choosing a strategy for the implementation of the interval arithmetic roundings within the multiple precision package. There were two primary methods to be considered.

The first of these involved imbedding the roundings solely in the four basic operations of multiply, divide, add and subtract. The implementations of the built-in functions would not be concerned with rounding strategies but would let this responsibility lie with the four basic operations. The primary advantage of this approach was its simplicity. The only algorithms which would require extensive analysis were those of the basic arithmetic operations. The disadvantage of this strategy, which precluded its use, was that is was quite inefficient with respect to cpu time consumption.

The second implementation method involved imbedding the rounding strategy in all implemented functions and operations. The primary disadvantage was that extensive algorithm analysis had to be performed on all implemented functions and operations. Its advantage, which more than offset its disadvantage, was that one could take advantage of the algorithmic structure of the various operations and functions to produce minimally wide interval results in the most efficient manner possible.

The largest portion of implementation time was taken up with the analysis and modification of operations and functions provided by the basic multiple precision package to perform interval arithmetic. This extensive analysis proved valuable in that an error in the basic multiple precision package was uncovered. The addition routine was incorrectly performing addition on the last

digit of the operands. One modification which proved to save much more time than it cost was that the subroutines were altered so that, if one desired, its original function (i.e. multiple precision real) could be invoked. That is, the desired truncation strategy is passed to the routine. The truncation strategies available are: upward directed, downward directed and standard rounding. This greatly eased the implementation of the VPREAL and VPINTEGER data types as well as reducing the overall size of the package by sharing code.

Once it had been assured that the basic set of operations and functions performed correctly when invoked with the various rounding strategies, thought was given to the implementation of the interval arithmetic aspects of the operations and functions. The addition routine, for example, could perform addition with upward directed, downward directed or standard roundings but did not perform interval addition per se. An approach was to develop a set of intermediate subroutines which would control the interval aspects of the operation or function. Each operation and function was to have a corresponding subroutine which would direct the roundings to conform to interval arithmetic.

After completion of these two stages, the imbedding of rounding strategies and the creation of interval operation subroutines, the package was equivalent to the single precision real interval package. There was the exception that interval variables could be of arbitrary precision; however, all interval variables had to be

of the same precision.

With the completion of this stage of the project, work was begun on automating the translation of user source containing the variable precision interval data type to standard FORTRAN. Not only was this part of the overall plan, but it would also greatly facilitate the testing of the programs. The first step was the creation of an AUGMENT description deck which would enable AUGMENT to perform the translation automatically. This was followed by the implementation of a skeletal "virtual compiler" to aid in the translation by automatically attaching the AUGMENT description deck to the extended FORTRAN source.

The installation of the preprocessor RATFOR on the Multics system was done at this time. The decision was made to insert RATFOR into the package because it allows the use of extended control structures (i.e. DO WHILE, IF-THEN-ELSE, REPEAT-UNTIL, etc.) in FORTRAN programming. The addition of extended control structures strongly complements the extended data types. The use of the extended control structures allows the user to produce more readable, more reliable and better documented programs faster than with standard FORTRAN. This is a strong argument for its inclusion.

The final step in the implementation of the basic variable precision interval arithmetic package was to find a means of allowing different variables to have different precisions. This

was done by "tagging" the data with its precision and by creating a third level of subroutines for precision control.

The third level of subroutines developed for precision control performs such activities as resolving precision conflicts among operands and the extraction of the precision for use by the lower levels. It should be noted that the implementation of this third level of subroutines went very quickly and easily. The basic structure of each subroutine is virtually identical for all functions and operations.

With the completion of this final stage of the implementation of the basic variable precision interval package, work was begun on the implementation of the SEPAFOR "virtual compiler" which would translate the user program with its extended data types and control structures to standard FORTRAN.

The SEPAFOR virtual compiler was constructed using PL/1 with calls to various Multics operating system modules. The function of the SEPAFOR virtual compiler is to take the user program written in SEPAFOR and map the functionality down to Multics object code. This is done by first passing the user's SEPAFOR source program through the RATFOR preprocessor which translates the extended control structures into standard FORTRAN still containing the extended data types. SEPAFOR dynamically generates an AUGMENT description deck using information obtained from the user's SEPAFOR source program. SEPAFOR takes the RATFORed source

and attaches the generated AUGMENT description deck producing an AUGMENT source deck. The AUGMENT source deck is then passed through AUGMENT to produce a standard FORTRAN source program. SEPAFOR then inserts into the standard FORTRAN source program initializations required by the variable precision interval arithmetic package. The final act of SEPAFOR is to invoke the FORTRAN compiler to produce Multics object code.

At this stage of the project, we had a useable subroutine package which would allow the user to write programs containing the extended data type VPINTERVAL. Each variable of this data type could be of arbitrary precision. That is, the variable COUNT could be of precision 20 while the variable STEP could be of precision 400. The entire process of translation of the user's SEPAFOR source program to Multics object code was automated so that no interaction was required from the user during the translation process. For some period of time after the completion of this stage of the project, the package was extensively tested to ensure that all was in working order and to minimize any problems that might appear at a later date. What remained of the implementation of SEPAFOR was the inclusion of the rest of the extended data types as outlined in the proposal.

4. Addition of Other Variable Precision Data Types

It was decided to implement VPREAL first as it was the simplest

of the extended data types to implement and would provide added experience for easing the implementation of the remaining extended data types. As has been mentioned before, the basic multiple precision package was an implementation of multiple precision real. The complete implementation of VPREAL required the creation of a set of intermediate subroutines for precision control. As it turned out, this set of subroutines is virtually identical to the precision control subroutines of VPINTERVAL. The exceptions being that the VPINTERVAL precision control subroutines call the interval arithmetic subroutines while the VPREAL precision control subroutines call the basic multiple precision routines directly and indicate the standard truncation rather than the directed roundings.

When it was realized that the precision control subroutines for VPREAL would be almost identical to the precision control routines for VPINTERVAL, the implementation of the VPREAL precision control subroutines went quickly. The implementation entailed text editing on the precision control routines of VPINTERVAL to produce the VPREAL precision control subroutines. The changes made were those outlined above.

A set of tests was again run on the VPINTERVAL portion of the package. This testing consisted of producing values with the VPINTERVAL data type. The same procedures were run again but with the VPREAL data type with a higher precision. A comparison was made between the two values. Theoretically, if nothing were

changed between the two runs except for the data type exchange and the higher precision, then the VPREAL values should always be contained in the interval produced with the VPINTERVAL data type. Such was the case with all tests run. This indicated that the modifications made to the basic multiple precision package in the hopes of minimizing the widths of the intervals produced appeared to be correct and proper.

With the completion of the implementation of VPREAL and its associated process of testing, attention was turned to the implementation of VPINTEGER. Since the basic multiple precision package produced only real values, integer arithmetic had to be simulated. To correctly simulate integer arithmetic, integer overflow had to be detected and truncation of fractional values had to be done properly.

There were two choices for the placement of the integer arithmetic simulation. The first of these was, like the interval arithmetic truncations, in the body of the basic multiple precision package. This would have entailed analyzing the routines again to determine which modifications were needed to produce only integer values and detection of integer overflow. It would also have raised the complexity of the modified routines. Considering the time it took for the interval arithmetic analysis, this was an unacceptable approach. The second approach was to go ahead and allow full computation of real values and perform checks on the results returned and was the approach

taken.

The precision control subroutines were then to be almost identical to the precision control subroutines of VPREAL. The exception being that before returning to the calling subroutine a call would be made to a subroutine which would check the value for integer overflow and truncate the fractional part. This would ensure that a proper integer value would be returned and that no integer overflow had occurred. The detection of integer overflow was simple in that all that had to be checked was the exponent. If the exponent was larger than the precision this indicated that integer overflow had occurred. Truncation was a matter of zeroing out the fractional part.

The remaining data type to be implemented was VPCOMPLEX. This proved somewhat more difficult to implement than VPREAL and VPINTEGER in that, like VPINTERVAL, a series of intermediate subroutines had to be written to perform complex arithmetic. The implementation was eased by the fact that standard truncation was to be used by VPCOMPLEX. The implementation of VPCOMPLEX did not take long as considerable experience had been gained in the manipulation of the base representation. Also, there was no need to consider the representation of the VPCOMPLEX data type; the representation used for the VPINTERVAL data type was quite adequate. The operations to be performed on the real and imaginary parts of VPCOMPLEX were similar to those performed on the left and right endpoints of VPINTERVAL. Thus, using the

intermediate subroutines of VPINTERVAL as a model, the writing of the intermediate subroutines for VPCOMPLEX was completed very quickly.

Concurrent with the implementation of each data type was the development of AUGMENT description decks to make possible the automated translation of the data types into standard FORTRAN. The use of the virtual compiler for each data type consisted of simply substituting the appropriate AUGMENT description deck into the translation process. At this stage of development we had four separate data types that could be used only in solitary. It was desired that all four should be able to be used simultaneously in the same program. To accomplish this there were two things that had to be done. A comprehensive AUGMENT description deck had to be developed. This AUGMENT description deck not only had to detail the particulars of each data type's conversion into standard FORTRAN, but had to detail how interactions among the different data types were to be carried out. The second task to be accomplished was the creation of a set of conversion routines for handling the interactions among the four extended data types as well as the standard FORTRAN data types.

The remaining tasks were carried out in short order. The creation of the comprehensive AUGMENT description deck was done by combining the description decks that had already been written. All that needed to be done to complete the comprehensive description deck was the addition of conversion information that

was needed for handling the interactions among the various data types. The writing of the conversion routines was made simple in that each data type's base representation was the same. For example, conversion from VPINTEGER to VPREAL was the equivalent of simple assignment. The only conversion which was of any trouble was the conversion from VPINTERVAL to the other data types; this entailed the writing of a subroutine for the taking of the midpoint of the interval. Once these two tasks were completed the package was ready for installation at WES.

5. Installation of SEPAFOR at WES

The transporting of SEPAFOR to WES consisted of two parts. First, the variable precision arithmetic package had to be delivered and installed on the G635 computer at WES. The delivery was made by magnetic tape. Second, the SEPAFOR virtual compiler had to be designed and written for compatibility with the G675 GECOS operating system.

The variable precision arithmetic package had been designed with high portability as one of its goals. Much care was taken in the writing of the variable precision arithmetic package to ensure that all constructs used were portable. To aid in this endeavor, PFORT was obtained from Bell Labs [11]. PFORT checks a FORTRAN program for compliance with a subset of ANSI standard FORTRAN. If a program is PFORT compatible then it should be compatible with

the majority of FORTRAN compilers in use today. This proved to be the case with the G635's FORTRAN compiler. The almost 13,000 lines of code compiled on the G635 without generating a single diagnostic!

Implementing the SEPAFOR virtual compiler proved to be much harder. Before transporting the system to WFS, as much of the Multics SEPAFOR virtual compiler as was possible was rewritten in FORTRAN. This included the AUGMENT description deck generator and the portion of SEPAFOR which inserts the initializing information. Upon arrival at WFS, the workings of SEPAFOR were explained and work was begun on its implementation. This implementation entailed the conversion of the PL/1-Multics portions of SEPAFOR which were not convertible to FORTRAN into G635 JCL. These portions included process control and file attachment for the various preprocessors. SEPAFOR was installed on the G635 system in four days with only two minor problems occurring. Since the package is integer arithmetic based, computations involving the extended data types should produce identical results on all computer systems on which they are executed. This is in sharp contrast to floating point based packages which, it seems, are almost guaranteed to produce different results on different computer systems.

References

[1] Ladner, T. D. and Yohe, J. M., "An interval arithmetic package for the UNIVAC 1108," The University of Wisconsin, Mathematics Research Center, Technical Summary Report No. 1055, May, 1970.

[2] Yohe, J. M., "Best possible floating point arithmetic," The University of Wisconsin, Mathematics Research Center, Technical Summary Report No. 1054, March, 1970.

[3] Binstock, W., Hawkes, J. and Hsu, N., "An interval input/output package for the UNIVAC 1108," The University of Wisconsin, Mathematics Research Center, Technical Summary Report No. 1212, September, 1973.

[4] Crary, F. D., "The AUGMENT precompiler, I. User information," The University of Wisconsin, Mathematics Research Center, Technical Summary Report No. 1469, December, 1974.

[5] Crary, F. D., "The AUGMENT precompiler, II. Technical documentation," The University of Wisconsin, Mathematics Research Center, Technical Summary Report No. 1470, October, 1975.

[6] Reuter Eric K., Jeter John P., Anderson J. W. and Shriver Bruce D. "Some Experiments Using Interval Arithmetic", Computer Science Department Report No. 78-7-1, University of Southwestern Louisiana, Lafayette, Louisiana, October, 1978.

[7] Podlaska-Lando, S. and Reuter Eric K., "Implementation and Evaluation of Interval Arithmetic Software, Report 2: The Honeywell Multics System", Technical Report No. 0-79-1, Office, Chief of Engineers, U. S. Army, Washington D. C. April 1979.

[8] Reuter Eric K., Jeter John P., Anderson J. W. and Shriver Bruce D. "A 56 Decimal Digit Implementation of an Interval Arithmetic Package on the Multics System", Computer Science Department Report No. 77-7-1, University of Southwestern Louisiana, Lafayette, Louisiana, September, 1977.

[9]. Software Tools, Kernighan, B. and Plauger, P., Addison-Wesley Publishing Company, Reading, Massachusetts, 1976.

[10] Brent, R. P., "A FORTRAN multiple-precision arithmetic package," Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, May, 1976.

[12] Ryder, B. G., "The PFORT Verifier", Software Practice and Experience, Vol. 4, 1974.

REFERENCES

A VARIABLE PRECISION INTERVAL DATA TYPE EXTENSION TO FORTRAN

A Project Report

Presented to

The Faculty and Graduate School

of the

University of Southwestern Louisiana

In Partial Fulfillment

of the Requirements for Degree

Master of Science

John P. Jeter

July, 1979

* This work was supported in part by the U. S. Army Corps of Engineers under grants DAAG29-78-G-0068 and DRXO-MA-15169-M

1. Introduction

The specific objectives of this project are:

1) to design a variable precision interval data type which would be imbedded in FORTRAN using the AUGMENT [4,5] preprocessor.

2) to implement R. P. Brent's multiple precision package [10], incorporating modifications to support interval arithmetic.

3) to implement the RATFOR preprocessor [9] and to design an AUGMENT description deck [4,5] incorporating them into the variable precision interval data type package. This would allow users to write FORTRAN programs containing the variable precision interval data type in a structured manner.

4) to perform an evaluation of the multilevel interpretation process that was used to implement structured variable precision interval FORTRAN.

The first section of this document gives an introduction to interval arithmetic that is helpful in understanding the properties of variable precision interval arithmetic. Further details concerning interval arithmetic are given in Appendix D. The second section describes the variable precision interval data type, its supporting arithmetic package and the translator which translates a user's "extended FORTRAN" program into standard FORTRAN. Appendices A and B give additional details concerning the RATFOR and AUGMENT preprocessors which are used to carry out portions of this translation. The last section in this report contains the summary of the project and the results of several program tests. In this section the variable precision

interval FORTRAN version of each test program is compared with other versions of the same algorithm using previously implemented arithmetics. An evaluation of the multilevel interpretation process is also presented. A user's guide to structured variable precision interval FORTRAN is included as an aid to the user writing structured variable precision interval FORTRAN programs.

2. General Considerations on the Computer Implementation of Interval Arithmetic

The finite precision arithmetic used on computers is an approximation to the real number system. In interval arithmetic, real numbers are approximated by intervals which contain the number. A brief introduction to interval arithmetic is given in Appendix D. Since the finite precision arithmetic used on computers is an approximation to the real number system, there are many intervals whose endpoints are not representable with a finite precision arithmetic. In this case the endpoints of the interval have to be approximated by the floating point system. This computer approximation of the real-valued intervals is represented as a pair of machine representable numbers stored in consecutive storage locations. The first number will be the lower bound of the machine approximation to the interval, referred to as the left endpoint, while the second number will be the upper bound, referred to as the right endpoint.

In order to obtain the smallest computer representable interval for the result of arithmetic operations on finite precision intervals, directed roundings on the computer arithmetic operations must be defined. Assume x is a real number and M1 and M2 are two machine representable numbers such that $M1 \leq x \leq M2$ and M1 and M2 are either equal or consecutive. Define Rd to be the downward directed rounding such that $Rd(x) = M1$. Define Ru to be the upward directed such that $Ru(x) = M2$. M1 and M2 will

be the machine representable numbers that are respectively the greatest lower bound and the least upper bound for the real number x. If x is a machine representable number, then Ru(x) = Rd(x) = x.

Algorithms for performing the machine arithmetic operations with directed roundings can be found in Yohe [2]. In general a op b, where a and b are machine representable numbers and op is one of the machine arithmetic operations, is not a machine representable number and must be rounded into a machine representable number. Directed roundings are used to compute the endpoints of the resultant interval for a particular arithmetic operation performed on two intervals. A downward directed rounding is performed to provide left endpoint and an upward directed rounding is performed to provide the right endpoint.

For example finite precision interval addition is defined as,

$$[a,b] + [c,d] = [a+c,b+d]$$

where a, b, c, and d are machine representable numbers. The computer approximation to the resultant interval is defined as follows:

$$[a,b] \oplus [c,d] = [Rd(a \oplus c), Ru(b \oplus d)]$$

where $\oplus$ is the machine addition operation.

Since the range of machine representable numbers is bounded, situations might occur during finite precision operations in

which these bounds are exceeded. If the finite precision number becomes too small, underflow has occurred. If the finite precision number becomes too large, overflow has occurred. If underflow occurs, then the true result is between zero and the smallest positive or negative representable number. In this case a directed rounding can give a valid bound. In the case of overflow, if rounding away from zero is wanted, then there is no machine representable number which can be used as a correct bound. This type of error condition, or fault, is known as an infinity fault.

2.1 Interval_Valued_Functions

A real-valued function, f, which is defined and continuous on an interval [a,b] can be extended to an interval-valued function, F, of an interval variable by defining

$$F([a,b]) = \{f(x) : x \in [a,b]\}.$$

When f is evaluated on a digital computer using machine representable approximations to the real numbers, a computer approximation, f', to f results. If F([a,b]) is an interval valued function of an interval (where a and b are machine representable numbers), then the computer approximation, F'([a,b]) is defined as an interval that contains F([a,b]).

Assume f' is the computer approximation of a real valued function f and f is monotonic increasing on [a,b]. Then

$$F'([a,b]) = [Rd(f'(a)),Ru(f'(b))]$$

where Rd is a downward directed rounding into a machine representable number such that $Rd(f'(a)) \leq f(a)$ and Ru is an upward directed rounding into a machine representable number such that $Ru(f'(b)) \geq f(b)$. Ideally we would like $Rd(f'(a))$ to be the largest machine representable number such that $Rd(f'(a)) \leq f(a)$ (i.e., a greatest lower bound) and $Ru(f'(b))$ to be the smallest machine representable number such that $Ru(f'(b)) \geq f(b)$ (i.e. a least upper bound).

If f is monotonic decreasing on [a,b], then

$$F'([a,b]) = [Rd(f'(b)),Ru(f'(a))]$$

If f is not monotonic on [a,b], then the interval [a,b] can be divided into disjoint subintervals; X'(i), i = 1,2,3,...,n; where the endpoints of each X'(i) are machine representable numbers and U X(i) contains all the machine representable numbers in the interval [a,b] and f is monotonic on each X'(i). In this case F'([a,b]) = U F (X'(i)). It should be noted that, in practice, this partitioning is performed only for the functions supplied by the support structure. If, for example, the user wished to evaluate the polynomial x**2 - x over the interval [.5±e] where is e is very small then the correct bounds may not be formed. However, the polynomial is broken down into subexpressions before evaluation; each of these subexpressions is evaluated with correct bounding since each is monotonic over the interval.

It may not be possible, due to algorithmic inadequacies or to accumulated roundings, to obtain the best bounds for the result of the computer approximation to the function f. The problem will be illustrated in the next section when describing the interval counterparts of the Multics basic external functions.

3. Structured Variable Precision Interval FORTRAN

The variable precision interval data type is patterned after the single precision interval data type previously implemented at University of Southwestern Louisiana [8] and at the Mathematics Research Center of the University of Wisconsin [2,11]. The computer representation of the single precision interval data type consists of a two element single precision floating point FORTRAN array. The first element in the array is the left endpoint while the second element is the right endpoint. The interval operations provided include the basic FORTRAN operations and supplied functions. These operations are supported at their lowest level by machine dependent procedures which perform the required directed roundings.

The variable precision interval data type's implementation is based upon R. P. Brent's Multiple Precision Package which has been modified to perform the necessary roundings for interval arithmetic. The implemented supporting package is highly portable. All modules have been successfully passed through the PFORT FORTRAN verifier[12]. PFORT is a subset of ANSI FORTRAN which should be compatible with the great majority of FORTRAN compilers. A good part of the process of translation from variable precision interval FORTRAN to standard FORTRAN was automated to make the task simpler for the user. The following subsections describe in further detail the variable precision

interval data type and the underlying supporting package as well as the translation process.

3.1 The Variable Precision Interval Data Type

The variable precision, interval data type is operationally the same as the single precision interval data type implemented earlier [8]. The difference lies in their basic machine representation. While the single precision interval data type is represented as two single precision real numbers, Figure 1, the variable precision, interval is represented as two single row integer arrays, or vectors, Figure 2. The variable precision, interval data type does, however, allow the user to specify the precision of each finite precision interval variable.

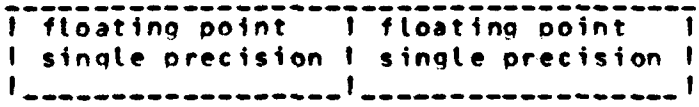


FIGURE 1

Single precision interval data type representation

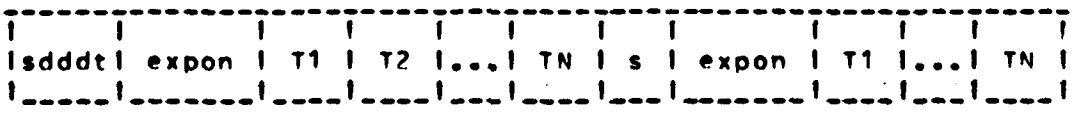


FIGURE 2

Multiple precision interval data type representation (sdddt = sign(0,-1 or +1) concatenated with the precision concatenated with a temporary variable indicator (1 for temporary, 0 for non-temporary) expon = exponent (to base b) T1 = digit (in base b) s = sign (0, -1 or +1))

As an example of the use of the variable precision, interval data type, suppose that a program written with variable precision, interval variables produces satisfactory results during most of the computation with a precision T1. Let us assume it executes a section of computation which has a data sensitivity; that is, for certain input data the algorithm produces results which are incorrect or, using interval arithmetic, the results have undesirably large interval widths. Using the variable precision, interval data type the user is able to specify that the computationally demanding section of code be performed with a precision T2 (T2>T1) which would be sufficiently high to produce satisfactory results. Conversely, for a less demanding computation the user may specify a lower precision. Thus, the variable precision, interval data type allows the user to tailor the precision of the interval variables (and subsequently the computational overhead) to the computational needs of the algorithm.

Structured, variable precision, interval FORTRAN has been designed so as to allow the user great freedom in the mixing of data types. Computations involving integer, real and variable precision, interval variables are allowed. The precision of interval operands are determined at runtime. The operations are performed with the same precision as the operand with the highest precision. The result is then converted to the precision of the

variable that is to contain the result, called the target. In cases where the target is an intermediate result of a computation the precision of the result is retained. Intermediate results retain the highest precision encountered during each computation. Further computations with an intermediate result are treated as above, using the highest precision required of the operands.

As a clarifying example consider the computation

$$Y = X*Z + A*Y + B + C$$

where X, Z, A, Y and B are variable precision, interval variables with precisions 25, 50, 75, 15 and 50 respectively and C is a real variable. The computation is performed as shown in Figure 3.

operation	precision	target
X * Z	50	temp1 (with precision 50)
A * Y	75	temp2 (with precision 75)
temp1 * temp2	75	temp1 (with precision 75)
temp1 + B	75	temp1 (with precision 75)
convert C to interval	10	temp2 (with precision 10)
temp1 + temp2	75	y (with precision 15)

FIGURE 3

Execution record of the FORTRAN statement Y = X*Z + A*Y + B + C displaying precision of the operation and the precision to which the result is converted

For a different perspective of the example let Y be a real variable and B an integer. Then the execution record would then

be that of Figure 4.

operation	precision	target
X * Z	50	temp1 (with precision 50)
convert B to interval	20	temp2 (with precision 20)
A * temp2	75	temp2 (with precision 75)
temp1 * temp2	75	temp1 (with precision 75)
temp1 + B	75	temp1 (with precision 75)
convert C to interval	10	temp2 (with precision 10)
temp1 + temp2	75	temp1 (with precision 75)
convert temp1 to real	75	y (in single precision)

FIGURE 4

Execution record of the FORTRAN statement Y = X*Z + A*Y + B + C with Y changed to a real variable and B changed to an integer displaying precision of the operation and the precision to which the result is converted

3.2 Supporting Arithmetic Package

Operations involving the variable precision, interval data type are implemented via a series of calls to a supporting multilevel interpretive structure. This multilevel interpretive structure controls the precision under which the interval operations are carried out, performs the necessary conversions, executes the desired operations with the required roundings and takes care of the housekeeping involved with the variable precision, interval data type. The supporting structure is composed of three separate levels. The first level of the multilevel interpretive structure performs the precision adjustments that need to be made before the actual operations are performed. The second level controls the details of the interval operation, for example, which

endpoints to use in determining the result and the rounding strategy to be employed. The third level carries out the actual operation with the proper roundings. The package can be represented graphically as shown in Figure 5.

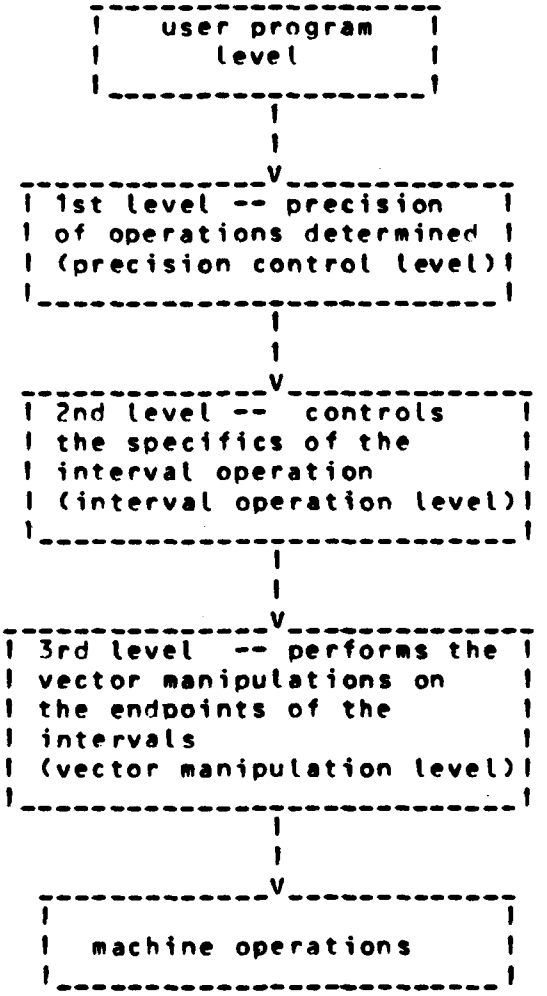


FIGURE 5

Graphic depiction of the various levels of the multilevel interpretive support structure

A demonstration of how the levels interact and the

responsibilities of each level can be made by following the computation C = A + B down through the different levels, where all variables are of type variable precision interval. The actions taken at each level are as follows:

1) User program level -- the assignment statement is translated into a call to the subroutine MVADD

```
CALL MVADD (A, B, C)
```

2) Precision control level -- MVADD inspects the operands A and B determining which has the larger precision. The smaller precision argument is converted to the larger precision by copying its value into a temporary workspace, named MVTEMP, kept explicitly for that purpose. The precision for which the computation will be performed, kept in an external location, is then set to the precision of the larger precision operand. A call to the interval operation level is then performed by executing either

```
CALL MXADD (MVTEMP, MVB, MVTEMP)  or
CALL MXADD (MVA, MVTEMP, MVTEMP)
```

The target is then inspected. If the target is itself a temporary, an intermediate result, the result of the operation is retained. If it is not, adjustments are made on the precision of the result and the value assigned to the target.

3) Interval operations level -- MXADD makes two calls to the vector operation level to perform the operation of interval addition

```
CALL MPADD   (MXA(right),  MXB(right),
     MXC(right), upward directed rounding
     indicator)
CALL MPADD (MXA(left), MXB(left), MXC(left),
     downward directed rounding indicator)
```

4) Vector manipulation level -- MPADD performs the manipulation of addition which is, at this low level, an addition of two integer arrays. This addition is performed by calls to various other supportive routines at this level which carry out the actual machine operations of addition on the integer array which represents the endpoint of the variable precision, interval argument specified

earlier.

3.3 "Structured" FORTRAN

The variable precision, interval data type has been imbedded in "structured" FORTRAN. This was accomplished through the incorporation of RATFOR [9], a preprocessor for FORTRAN written in FORTRAN, into the supporting structure. The end result was the production of structured, variable precision, interval FORTRAN.

"Structured" FORTRAN allows the use of contemporary control structures such as DO-WHILE, IF-THEN-ELSE and REPEAT-UNTIL [9]. The primary purpose is to make FORTRAN a better programming language by permitting and encouraging the writing of readable and well-structured programs. This is done by providing the control structures that are unavailable in FORTRAN, and by improving the "cosmetics" of the language, similar to that done in FORTRAN '77.

The cosmetic aspects of RATFOR have been designed to make it concise and reasonably pleasing to the eye. It is free-form; statements may appear anywhere on an input line. Other additions also improve the readability of the language; for example, the use of the symbol ">" conveys the meaning of code more rapidly than the equivalent string of symbols ".GT.".

To show the advantages of RATFOR consider the construct

IF (condition) THEN (s1) ELSE (s2).

This construct is, for the most part, fairly easy to understand. IF the "condition" is true THEN statement "s1" is to be executed, ELSE statement "s2" is to be executed. However, this construct is rather awkward to express in FORTRAN.

As an example, suppose that if the value of the variable X were greater than or equal to 10.7, then X is to be divided by 18.3 and the counter KOUNT incremented by one. However, if the condition were false then the variable X is to be multiplied by 18.3 and the counter KOUNT decremented by one. One way of expressing this in FORTRAN as shown in Figure 6.

```
      IF (X.GE.10.7) GO TO 10
      X = X * 18.3
      KOUNT = KOUNT - 1
      GO TO 20
10    X = X / 18.3
      KOUNT = KOUNT + 1
20    CONTINUE
```

FIGURE 6

Example of FORTRAN code

On the other hand, the same logic could be expressed in RATFOR as shown in Figure 7.

```
IF (X >= 10.7)
  [
   X = X / 18.3
   KOUNT = KOUNT + 1
  ]
ELSE
  [
   X = X * 18.3
   KOUNT = KOUNT - 1]
```

FIGURE 7

RATFOR version of FORTRAN code in Figure 6

With these additional contemporary programming language constructs the programmer is able to produce a readable, structured program and have it translated automatically into standard FORTRAN. It is herein that RATFOR's greatest value lies. A highly readable, structured program is a program that is easier to develop, debug and modify. The time it takes a programmer assigned to modify an existent program to get the job done is directly proportional to the understandability of the code. By the automatic translation of these constructs into FORTRAN, the programmer is able to devote a larger portion of time to the development of ideas rather than their translation while being assured that this translation will be done correctly each and every time.

3.4 Translation of Structured Variable Precision Interval FORTRAN to FORTRAN

A "virtual" compiler was developed to compile the structured variable precision, interval FORTRAN source to the "object code" of the FORTRAN virtual machine. The virtual compiler allows the user to write "structured" FORTRAN programs which contain interval variables. The virtual compiler automatically performs the simple but laborious task of tending to the technical details of the translation. The virtual compiler program first passes the application's source program through the preprocessor RATFOR producing an intermediate form of the program. This intermediate program is then passed to AUGMENT after an AUGMENT description deck has been automatically attached. A FORTRAN version of the program is then produced by AUGMENT. This process is graphically depicted in Figure 8. A more detailed description of RATFOR and the AUGMENT description deck can be found in the appendices.

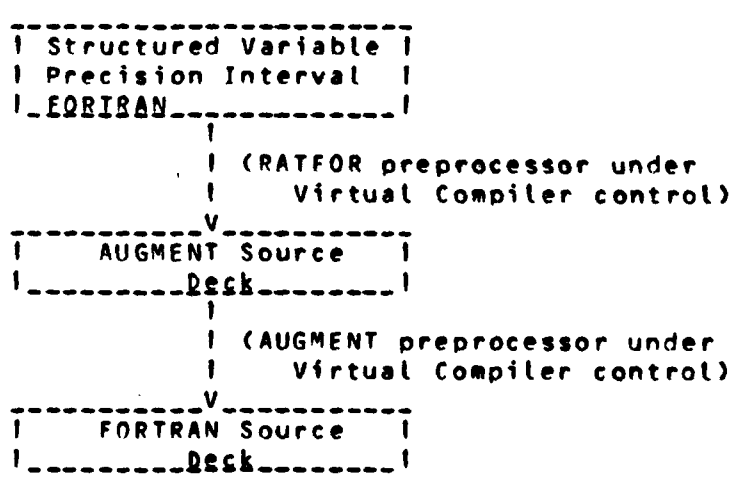


FIGURE 8

Depiction of translation process

4. Results_and_Summary

4.1 Results

The test programs used as a basis for the results consist of a heat transfer program and a program which computes the constant e. The latter was chose for its simplicity. It consists of a simple iterative loop with only the basic operations of addition, multiplication and division. One other reason for its inclusion is that the constant e is known to many decimal places and therefore some measure of the capabilities of the various arithmetics can be made. This algorithm also provides an example of the effects that extending loop termination factors can have on resource usage when extended precision is used. The algorithm used to compute the constant e is given is Figure 9.

```
/* initializations */
sum = 0; nfact = 1;
step = 1; i = 0;
error_factor = CONSTANT

/* main loop */
repeat
   sum = sum + step
   i = i + 1;
   nfact = nfact * i;
   step = 1/nfact
until (step <= error_factor);

/* output result */
put (sum);
```

FIGURE 9

Algorithm to compute the constant e

```
/* initializations */
k = 30; l = 10; c = .12; t0 = 70;
t1 = 500; ro = 7.1 * 62.3;
pi = 3.1415927;
get (theta);

/* loop once for each foot */
do feet = 1 to l;

   /* initializations for sub-loop */
   t = (k * theta)/(l**2 * ro * c);
   x = feet/l;  /* increment foot */
   sum = 0; count = 0; time = 0;

   /* loop to determine temperature */
   repeat
      count = count + 1;
      xsum = ((-1)**count/count) *
             exp((-count)**2 * pi**2 * t *
             sin(count*pi*x);
      sum = sum + xsum;
      if (abs(xsum) < error
        then time = time + 1;
      else time = 0;
   until (time = 2);

/* compute and output result */
t = t0 + (t1-t0) * (x + (2/pi)*sum);
put (t);
end;
```

FIGURE 10

Algorithm to compute the temperature of a pipe of length l at 1 foot intervals, where the ambient temperature is t0 degrees and the heat source is t1 degrees

The heat transfer program, an applications source program in use by the Mechanical Engineering Department of the University of Southwestern Louisiana, was chosen to present an algorithm which

possessed somewhat more sophistication. It is composed of a more complex looping structure. It also contains more complex operations, sine, exponential, conversion, etc. The wide range of operations will serve to provide a basis for a comparison of the arithmetics on a cost per digit basis. The algorithm which is used in the heat transfer program is presented in Figure 10.

4.1.1 Benchmarks with Comparisons to Previous Arithmetics Implemented at USL

The following results were produced on the Honeywell Multics system based on a Honeywell 68/80 two processor configuration. Two metrics are given for the results produced, cpu time and paging. It should be noted that the cpu time given is the virtual cpu time for the run. There has been observed some slightly irregular behavior in the system routine which supplies the virtual cpu time; therefore, these figures are not presented as the actual amount of cpu time consumed. It is felt that these figures are reliable enough for performing valid comparisons. The paging results presented are more a function of system load rather than amount of main memory usage. There is, however, some small but noticeable cost incurred during a page fault. Thus, a run with a great number of page faults will consume a greater amount of virtual cpu time than the same run with few page faults. This metric has major interest only to those intimately familiar with Multics.

4.1.1.1 Computation of the Constant e

Figure 11 presents the value of e to 25 decimal digits of precision. This value was acquired from a handbook of mathematical tables [7]. It should provide a basis for the comparison of the following results. Appendix H also contains a value for e which was obtained by executing a multiple precision version of the e computation algorithm with a precision of 1500 digits and rounding the result to 1000 digits. This value should be accurate to all 1000 digits. Source versions of the e program can be found in appendix I.

```
+-----------------------------------+
|                                   |
|  2.7182818284590452353602874      |
|                                   |
+-----------------------------------+
```

FIGURE 11

Value of e to 25 digits

The following two results were obtained from single precision and double precision FORTRAN versions of the program to compute the constant e. The single precision version, with an error factor of 1.0e-8, has a result which is accurate to 7 digits. The double precision version, with an error factor of 1.0e-15, has a result which is accurate to 19 digits. The double precision version produced results which were accurate to more than twice as many digits as the single precision version.

```
e IS EQUAL TO          2.71828178
COMPUTED IN  10 STEPS  ACCURACY = 7 DIGITS
CPU time =     0.024123 seconds; Page faults =          0
```

FIGURE 12

Results of single precision e computation
(ERROR = 1.0e-8)

```
e IS EQUAL TO       2.7182818284590452 35
COMPUTED IN  18 STEPS  ACCURACY = 19  DIGITS
CPU time =     0.024731 seconds; Page faults =          0
```

FIGURE 13

Results of double precision e computation
(ERROR = 1.0e-15)

The results presented in Figure 14 are those from the single precision interval version of the program. It has an error factor of 1.0e-8 with an accuracy of 7 digits. The interval width is quite small indicating that the algorithm is stable with respect to truncation error. Note that the single precision real, double precision real and the actual value are contained within the interval, which is quite desirable. The results did, however, consume more than an order of magnitude greater amount of cpu time than did the single precision real version. The price paid bought the greater trust in the results produced.

```
e IS EQUAL TO [ 2.71828172, 2.71828198 ]
COMPUTED IN  10 STEPS  ACCURACY = 7 DIGITS
CPU time =     0.473424 seconds; Page faults =          3
```

FIGURE 14

Results from single precision interval e computation
(ERROR = 1.0e-8)

The results presented in Figure 15 were produced from the 56-decimal digit interval version of the program with an error factor of 1.0e-50. The result is accurate to ?? digits. The execution incurred a cpu cost more than two orders of magnitude greater than that of the double precision real version and an order of magnitude greater than the single precision interval version.

```
e IS EQUAL TO
[ 2.71828182845904523536028747135266249775724709369995884,
  2.71828182845904523536028747135266249775724709369995885 ]
COMPUTED IN  42 STEPS  ACCURACY = 51 DIGITS
CPU time =     5.534146 seconds; Page faults =          7
```

FIGURE 15

Results from 56 decimal digit interval e computation
(ERROR = 1.0e-50)

There now follows a series of results produced by the variable precision interval version of the program. The first result is supplied for comparison to the 56 decimal digit arithmetic. The other results, all with error factors of 90% of the precision, provide for the determination of the relationship between amount of precision and accuracy of the results that is obtainable. They will also give some idea of the cpu costs incurred when extending the precision of a computation using variable precision interval arithmetic.

The results of Figure 16 are computed with 56 decimal digits of precision with an error factor of 1.0e-50. When compared to the results gained from the 56 decimal digit interval version with

the same error factor one finds that the variable precision interval version incurred a cpu time cost that was only slightly greater. The accuracy was the same as with the 56 decimal digit version.

```
e IS EQUAL TO
[ 2.71828182845904523536028747135266249775724709369995884 60,
  2.71828182845904523536028747135266249775724709369995884 65 ]
COMPUTED IN  42 STEPS   ACCURACY = 51 DIGITS

CPU time =     7.043885 seconds; Page faults =          12
```

FIGURE 16

Results from 56-digit variable precision interval e computation
(ERROR = 1.0e-50)

The next four results were computed with 100, 200, 500 and 1000 digits of precision respectively. The accuracy, decimal digits, was 91, 181, 452 and 899 respectively. As can be seen, in all cases the number of digits of accuracy closely matched the error factor.

```
e IS EQUAL TO
[ 2.71828182845904523536028747135266249775724709369995957 49
   66967627724076630353547594571382178402054876 1,
  2.71828182845904523536028747135266249775724709369995957 49
   66967627724076630353547594571382178402054882 3     ]
COMPUTED IN  65 STEPS  ACCURACY = 91 DIGITS

CPU time =    12.832085 seconds; Page faults =          14
```

FIGURE 17

Results from 100-digit variable precision interval e computation
(ERROR = 1.0e-90)

```
e IS EQUAL TO
[ 2.71828182845904523536028747135266249775724709369995957496
    69676277240766303535475945713821785251664274274663919320
    03059921817413596629043572900334295260595630738132328627
    94349076323325754074104071474386,
  2.71828182845904523536028747135266249775724709369995957496
    69676277240766303535475945713821785251664274274663919320
    03059921817413596629043572900334295260595630738132328627
    94349076323325754 074104071474494       ]
COMPUTED IN 111 STEPS  ACCURACY = 181 DIGITS

CPU time =    42.957277 seconds; Page faults =          44
```

FIGURE 18

Results from 200-digit variable precision interval e computation
(ERROR = 1.0e-180)

```
e IS EQUAL TO
[ 2.71828182845904523536028747135266249775724709369995957496
    69676277240766303535475945713821785251664274274663919320
    03059921817413596629043572900334295260595630738132328627
    94349076323382988075319525101901157383418793070215408914
    99348841675092447614606680822648001684774118537423454424
    37107539077744992069551702761838606261331384583000752044
    93382656029760673711320070932870912744374704723069697720
    93101416928368190255151086574637721112523897844250569536
    96760419207448518691461321006992866910259014501704840
  2.71828182845904523536028747135266249775724709369995957496
    69676277240766303535475945713821785251664274274663919320
    03059921817413596629043572900334295260595630738132328627
    94349076323382988075319525101901157383418793070215408914
    99348841675092447614606680822648001684774118537423454424
    37107539077744992069551702761838606261331384583000752044
    93382656029760673711320070932870912744374704723069697720
    93101416928368190255151086574637721112523897844250569536
    96760419207448518691461321006992866910259014501707140     ]
COMPUTED IN 233 STEPS  ACCURACY = 452 DIGITS

CPU time =   291.310798 seconds; Page faults =          48
```

FIGURE 19

Results from 500-digit variable precision interval e computation
(ERROR = 1.0e-450)

```
e IS EQUAL TO
[ 2.71828182845904523536028747135266249775724709369995957496
    69676277240766303535475945713821785251664274274663919320
    03059921817413596629043572900334295260595630738132328627
    94349076323382988075319525101901157383418793070215408914
    99348841675092447614606680822648001684774118537423454424
    37107539077744992069551702761838606261331384583000752044
    93382656029760673711320070932870912744374704723069697720
    93101416928368190255151086574637721112523897844250569536
    96770785449969967946864454905987931636889230098793127736
    17821542499922957635148220826989519366803318252886939849
    64651058209392398294887933203625094431173012381970684161
    40397019837679320683282376464804295311802328782509819455
    81530175671736133206981125099618188159304169035159888851
    93458072738667385894228792284998920868058257492796104841
    98443634632449684875602336248270419786232090021609902353
    04369941849146314093431738143640546253152096183690888707
    01599908889495337516730483941167953228780215021423391110
    13587395889105589592751351142817287815786 02
  2.71828182845904523536028747135266249775724709369995957496
    69676277240766303535475945713821785251664274274663919320
    03059921817413596629043572900334295260595630738132328627
    94349076323382988075319525101901157383418793070215408914
    99348841675092447614606680822648001684774118537423454424
    37107539077744992069551702761838606261331384583000752044
    93382656029760673711320070932870912744374704723069697720
    93101416928368190255151086574637721112523897844250569536
    96770785449969967946864454905987931636889230098793127736
    17821542499922957635148220826989519366803318252886939849
    64651058209392398294887933203625094431173012381970684161
    40397019837679320683282376464804295311802328782509819455
    81530175671736133206981125099618188159304169035159888851
    93458072738667385894228792284998920868058257492796104841
    98443634632449684875602336248270419786232090021609902353
    04369941849146314093431738143640546253152096183690888707
    01599908889495337516730483941167953228780215021423391110
    13587395889105589592751351142817287815790 11 ]
COMPUTED IN 412 STEPS  ACCURACY = 899 DIGITS

CPU time =  1574.385312 seconds; Page faults =          51
```

FIGURE 20

Results from 999-digit variable precision interval e computation
(ERROR = 1.0e-900)

To summarize the results of the comparison of the arithmetics -- the higher the precision the better the results that could be obtained and the more cpu time that was consumed. This was expected; not only was there more information (digits) to be processed, but the extra precision allowed the extension of the loop termination factor. The extension of the loop termination factor resulted in the loop being executed a greater number of times which in itself would account for a large increase in cpu time consumption (Figure 21). A summary table of the various run times for the different arithmetics is given in Figure 21. The cost of the arithmetic used must be weighed against its benefits. For this program the single precision version produced acceptable results; the double precision version produced excellent results. The single precision interval version did not bring to light any faults in the algorithm. Since the single precision interval version is much more sensitive than any of the other interval versions there was no justification for the use of the other versions other than as a point of comparison. The results of the extended precision interval versions did demonstrate that these arithmetics were capable of providing a high level of significance in those situations which warrant its use.

error	real	double	single int	56 int	VARINT cpu [digits]	number of liters	digits of acc
10-8	.02	**	.47	**	**	10	7
10-15	**	.02	**	**	**	18	19
10-50	**	**	**	5.5	7.0 [56]	42	51
10-90	**	**	**	**	12.8 [100]	65	91
10-180	**	**	**	**	43.0 [200]	111	181
10-450	**	**	**	**	291.3 [500]	233	452
10-900	**	**	**	**	1574.4 [999]	412	899

FIGURE 21

Table of run times (in seconds) of various arithmetics

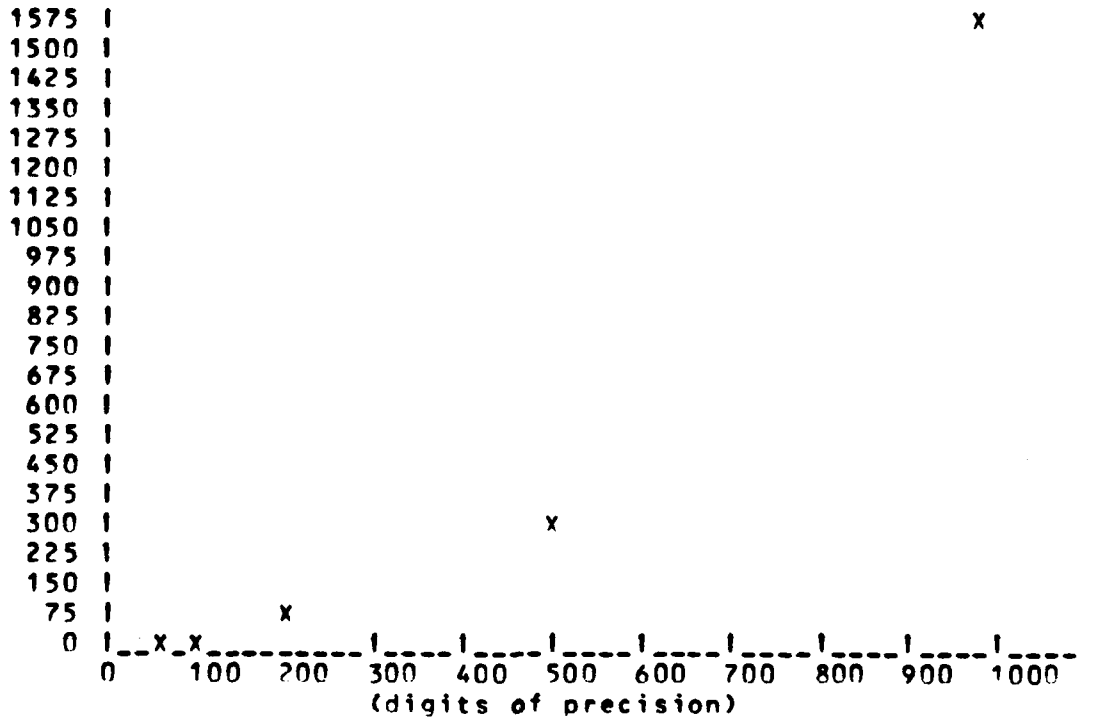


FIGURE 22

Plot of digits of precision versus cpu time consumed

4.1.1.2 The Heat Transfer Program

Exact results are not known for this program as was the case with the e computation; these results are presented primarily for comparisons of the various arithmetics on a cost per digit basis. Hence, all versions of the program use the same loop termination factors; there is no other difference among the various versions other than precision. The algorithm is also sufficiently complex to allow a demonstration of the effects that algorithm configuration can have upon cpu time consumption. The various versions of the algorithm are presented in Appendix J. Figure 24 displays the cpu timings for each of the runs along with the largest interval width produced by the different interval versions. For those interested the various results are to be found in appendix K. Figure 23 displays the output from the single precision real version as a sample output.

```
-------------------------------------------------
| the answer for  1 foot is          70.000000 |
| the answer for  2 feet is          70.000000 |
| the answer for  3 feet is          69.999996 |
| the answer for  4 feet is          70.000006 |
| the answer for  5 feet is          70.001100 |
| the answer for  6 feet is          70.072399 |
| the answer for  7 feet is          72.054155 |
| the answer for  8 feet is          95.780487 |
| the answer for  9 feet is         219.179291 |
| the answer for 10 feet is         500.000050 |
|-------------------------------------------------|
| CPU time = 0.178528 seconds; Page faults =   0 |
|-------------------------------------------------|
```

FIGURE 23

heat transfer -- single precision real version

version	largest interval width	cpu time consumption
single	**	.19
double	**	.20
single int	1.0e-4	6.87
56 int	1.0e-51	1812.56
VARINT 56	1.0e-53	325.10
VARINT 100	1.0e-96	725.97
VARINT 200	1.0e-197	2406.90
VARINT 500	1.0e-495	15505.37
VARINT 200 +	1.0e-17	2396.17
VARINT 200 ++	1.0e-15	2377.67

FIGURE 24

Table of run times (in seconds) of various arithmetics
(+ -- with output only produced at 20 digits precision)
(++ -- all invariant expressions removed from loops
and minimum necessary precision used)

The first four results presented in Figure 24 are those obtained with the previously implemented arithmetics. Notice that the 56 decimal digit interval version shows a somewhat anamolous amount of cpu consumption. This results from the fact that the 56 decimal digit interval support package relies upon the multiple precision package for the production of several supplied functions. Among these are the sine and exponential functions which partially form the main expression in the inner loop of the computation. The cpu time consumption of the 56 decimal digit interval version is strongly affected by this usage of the multiple precision routines. The result is that the cpu time consumption for this version is above what one would expect after

viewing the results of the e computation benchmark, which included extensions to the loop termination factors.

The next four results of Figure 24 are those obtained with the variable precision interval version of the heat transfer program with precisions of 56, 100, 200 and 500 respectively.

The next result shown in Figure 24 was partially computed with 20 digits of precision. The inner loop of the algorithm was still computed with 56 digits of precision, but the value to be output was produced with a precision of 20 digits. It was felt that this more closely reflected a reasonable number of digits output. That is, output to more than 20 digits or so of precision is not realistic; digits after the 20th would more than likely be ignored. The reduction of the only the output computation does not result in any large reduction in the total amount of cpu time consumed. It is, however, significant when one considers that the reduced precision operations account for only 60 operations out of more than 6000 total operations.

The last result shown in the table was produced with a fully optimized version of the heat transfer program. That is, optimized in the sense that all invariant expressions are removed from loops and constants are stored only to the precision necessary to maintain their integrity. There is a much more

substantial reduction in the cpu time consumption than was observed in the previous partially optimized version.

In summary, the writer of programs containing VARINTERVAL variables must remember that operations upon these variables to any substantial precision incurs a non-trivial amount of cpu time consumption. Thus previously acceptable algorithm configurations are not appropriate when implemented with variable precision interval variables. The optimization techniques used on the VARINTERVAL version of the algorithm would not have been worth the effort on the single precision version of the algorithm. Little savings can be realized in a run which consumes cpu time measured in the hundreths of a second. The rewards are, however, sufficiently great for one to apply what were previously trivial optimizations to programs containing the variable precision interval data type.

4.1.2 An Evaluation of the Multilevel Interpretation Process

The multilevel interpretive support structure consists of three levels. As has been mentioned previously, each level has its own set of clearly defined responsibilities. The first level controls the precision of the operation. The second level contains the logic for controlling the interval aspects of the operation. The third level performs the actual operation as a set of operations

on vectors. The multilevel interpretive support structure is to be evaluated at each level. This evaluation will be made by choosing one of the primitives provided by the supporting structure and tracing the interpretive process through each level. Appropriate comments will be made concerning the interactions of the various levels as well as identifying the salient features at each level.

The primitive chosen for this evaluation is MVPWR, the power function. A graphic display of the multilevel interpretive structure of MVPWR can be seen in Figure 25 which presents the calling sequences of the various levels of the primitive as a tree. At the top level is MVPWR which initiates a call to MXPWR at the second level. MXPWR is supported at the third level by a broad base of multiple precision package routines [Appendix C]. Consider, now, each level in turn.

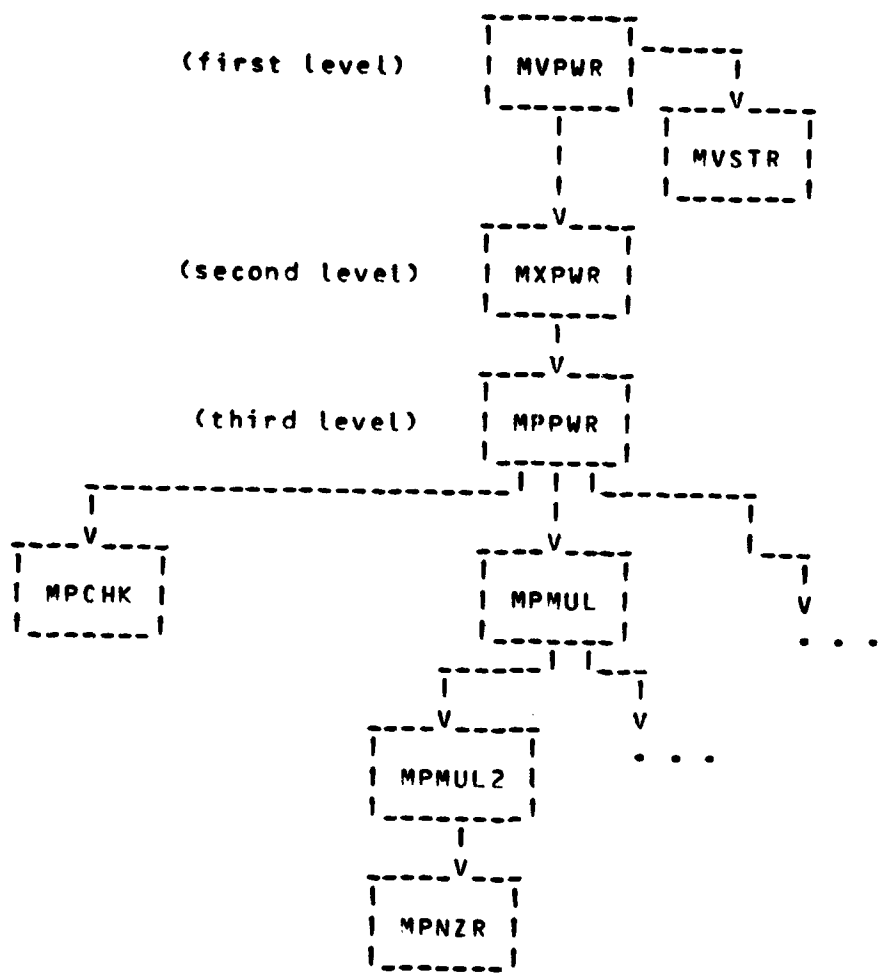


FIGURE 25

Tree diagram of the multilevel interpretive structure of the power function

The first level, MVPWR, controls the precision at which the operation is to be performed. The first action taken at this level is the extraction of the precision of the operand. The precision at which the operation is to be performed is that of the operand. The precision at which the operation is to be

performed is set by assigning the precision of the operand to the variable T, located in common. The common variable T is referenced at the lower levels to determine the precision of the operation. Once the precision has been set, the format of the operand, Figure 2, is converted to that used at the lower levels, Figure 49. The only difference is the insertion of four extra digits of information in the first word of the interval variable. The first three of these extra digits contain the precision. The last digit is a temporary variable indicator used to signify whether or not the variable contains an intermediate result of a computation and as such may have its precision altered. This sequence is depicted in the code section in Figure 26.

```
      C STATEMENT FUNCTION FOR THE EXTRACTION OF THE COMBINED
      C  PRECISION AND TYPE
            EXTRACT(WORD1) = IABS(WORD1 - (WORD1/10000)*10000)
      C
      C PICK OFF THE TYPE AND THE PRECISION OF THE ARGUMENT
            WORD1ARG = ARG(1)
            PREC = EXTRACT(WORD1ARG)/10
      C
      C SET THE PRECISION FOR THE LOWER LEVELS
             T = PREC
      C
      C CONVERT THE FIRST WORD TO MP FORMAT
             ARG(1) = ARG(1)/10000
```

FIGURE 26

Code section from MVPWR performing argument preparation

The operand is then passed to the second level with a call to MXPWR. The result is returned in the temporary work space TEMP. The operand is restored to its previous format. TEMP, which is

returned in the lower level format, is converted to the first level format using the precision of the operand. The value of TEMP is then assigned to the target with a call to MVSTR. MVSTR inspects the temporary variable indicator of the target. If the target is a temporary variable then the contents of TEMP and the precision information are copied to the target. If the target is not a temporary variable then adjustments to the value of TEMP must be made before the assignment is carried out. These adjustments are of the form of either truncation or the filling in of unused digits of the target with zeros, depending upon whether the precision of TEMP is greater than or less than that of the result. The target is then returned to the user program. This sequence is depicted in Figure 27.

```
      C CALL THE SECOND LEVEL TO PERFORM THE OPERATION
            CALL MXPWR (ARG(1), N, TEMP)
      C
      C RESTORE THE PARTICULARS OF THE FIRST WORD OF THE
      C ARGUMENT AND THE TEMPORARY TEMP GETS THE PRECISION OF ARG.
            TEMP(1) = ISIGN(IABS(TEMP(1))*10000 +
     &                      EXTRACT(WORD1ARG), TEMP(1))
            ARG(1) = WORD1ARG
      C
      C MAKE THE ASSIGNMENT TO RESULT
            CALL MVSTR (TEMP, RESULT)
      C
            RETURN
```

FIGURE 27

Code section of MVPWR for result preparation

This first level incurs the least overhead of the three levels. The major portion of the overhead is the vector copy operation

performed during the assignment of the value of TEMP to the target. The overhead of precision determination and format conversion is quite small in comparison with the overall resource usage unless the precision is unusually small.

The second level, MXPWR, contains the logic for performing the interval aspects of the power operation. The primary function of this level is the performance of a case analysis to determine which endpoints are to be used in the production of the result. The case analysis is necessary to ensure that the result is a valid interval. In general, the relation [a1,a2]**n = [a1**n,a2**n] does not hold. For example, [1,2]**(-2) = [2**(-2),1**(-2)] = [.25, 1]. Additional information pertaining to the interval operations may be found in Appendix D. The case analysis for MXPWR is displayed in Figure 28. The endpoints of the interval operand are passed singly to the third level with an indicator which stipulates the proper rounding strategy. For an interval [A1,A2], a power N and a target [Z1,Z2] with the above values the two calls would be

```
CALL MPPWR (A1, N, Z2, 2)
CALL MPPWR (A2, N, Z1, 1)
```

The value 1 is used to indicate a downward directed rounding, while the value 2 indicates an upward directed rounding.

```
CASE 1. the POWER is even and positive and the
          OPERAND is positive
        RESULT(left)  = OPERAND(left)  ** POWER
        RESULT(right) = OPERAND(right) ** POWER
CASE 2. the POWER is even and positive and the
          OPERAND is negative
        RESULT(left)  = OPERAND(right) ** POWER
        RESULT(right) = OPERAND(left)  ** POWER
CASE 3. the POWER is even and negative and the
          OPERAND is positive
        RESULT(left)  = OPERAND(right) ** POWER
        RESULT(right) = OPERAND(left)  ** POWER
CASE 4. the POWER is even and negative and the
          OPERAND is negative
        RESULT(left)  = OPERAND(left)  ** POWER
        RESULT(right) = OPERAND(right) ** POWER
CASE 5. the POWER is odd and positive
        RESULT(left)  = OPERAND(left)  ** POWER
        RESULT(right) = OPERAND(right) ** POWER
CASE 6. the POWER is odd and negative
        RESULT(left)  = OPERAND(right) ** POWER
        RESULT(right) = OPERAND(left)  ** POWER
CASE 7. the POWER is even and positive and the
          OPERAND contains zero and the absolute
          value of the right endpoint is greater than
          the absolute value of the left endpoint
        RESULT(left)  = 0
        RESULT(right) = OPERAND(right) ** POWER
CASE 8. the POWER is even and positive and the
          OPERAND contains zero and the absolute
          value of the right endpoint is less than
          the absolute value of the left endpoint
        RESULT(left)  = 0
        RESULT(right) = OPERAND(left)  ** POWER
CASE 9. the POWER is even and negative and the
          OPERAND contains zero
        DIVISION BY ZERO ERROR
```

FIGURE 28

Case analysis for interval power function

The overhead incurred at this level is a function of the operands. The case analysis of MXPWR serves as an example. The determination of cases one through six requires very little

computation. The determination of these cases involves only an inspection of the word containing the sign rather than the vector as a whole. For cases seven through ten, however, the determination of the case requires a comparison between the two endpoints of the interval operand. This can entail a digit-by-digit comparison of the endpoints. Naturally, the greater the precision, the greater the overhead that will be incurred during the comparison.

The third level, MPPWR, performs the operations on the vectors representing the endpoints of the interval variable. At this level the interval endpoints are no longer considered as forming one entity, but are treated as separate operands. This level receives from the second level a single endpoint along with an indicator which stipulates the truncation strategy to be used. An outline of the algorithm used in MPPWR is presented in Figure 29. The algorithm forms the power by performing successive multiplications. For negative powers the reciprocal of the argument is first formed with a call to MPREC. The multiplications are performed by MPMUL.

```
# X**0 = 1
IF N = 0
  [ RESULT = 1
    RETURN ]

# SETUP FOR MULTIPLICATIVE LOOP
IF N < 0
  [ TEMP = 1/X ]
ELSE
  [ TEMP = X ]
RESULT = 1

# MULTIPLICATIVE LOOP
REPEAT
  [ IF (N IS ODD) RESULT = RESULT * TEMP
    TEMP = TEMP ** 2
    N = N/2 ]
UNTIL (N = 0)
```

FIGURE 29

Algorithm for MPPWR (X**N)

MPMUL performs the multiplication between two multiple precision numbers as one would multiply in longhand. That is, operand one is multiplied by the last digit of operand two forming the result in a work space in common. Then operand one is multiplied by the second to the last digit in operand two adding the result to the value in the workspace in common after shifting left one digit. The multiplication of operand one by a digit in operand two is performed with a call to MPMLP. After the result has been formed in the workspace in common, it is normalized and roundings are performed by MPNZR. An outline of the algorithm of MPMUL is presented in Figure 30.

```
# COMPUTE THE SIGN AND EXPONENT
#   OF THE RESULT
RESULT_SIGN = SIGN(X) * SIGN(Y)
RESULT_EXPONENT = EXPONENT(X)+EXPONENT(Y)

# MULTIPLICATIVE LOOP
DO I = 1 TO PRECISION
  [ # MULTIPLY VECTOR BY SCALAR
    TEMP = TEMP + SHIFT_LEFT_I(X *
         ITH_DIGIT(Y))
    PROPAGATE CARRIES ]

NORMALIZE AND ROUND RESULT
Z = RESULT
```

FIGURE 30

Algorithm for MPMUL(X, Y, Z)

MPNZR performs the normalization and rounding of the result in the workspace in common and assigns the value to the target of the power operation. The normalization is performed as one would expect; the digits are shifted left until the first digit is a non-zero digit. Adjustments are then made to the exponent to reflect the shift. The rounding is somewhat more involved. The rounding strategy used to round the result is that stipulated by the second level, either a downward directed rounding or an upward directed rounding. The results of interval operations at this level are carried to twice the precision with four guard digits to minimize the loss of information due to finite precision representation of real results. MPNZR makes use of these additional digits of the result in carrying out the roundings. MPNZR inspects the sign of the result to determine

the necessary action to be performed to carry out the specified rounding strategy. For a negative value, an upward directed rounding requires a simple truncation of the result to the necessary precision. For example, for three digits of precision the value -.3339999999 would be rounded to -.333. For a positive value an upward directed rounding requires the inspection of the additional digits carried in the result for non-zero values. If any non-zero values are found then the result is truncated to the necessary precision and one is added to the last digit. For example, for three digits of precision the value .3330000001 would be rounded to .334. The actions performed for a downward directed rounding are the reverse of those performed for the upward directed rounding. A summary of these actions is presented in Figure 31.

rounding	value	action
upward	positive	add one if non-zero additional digits
upward	negative	truncate
downward	positive	truncate
downward	negative	add one if non-zero additional digits

FIGURE 31

Summary of rounding actions

Nearly all of the overhead associated with the variable precision interval data type is incurred at this level. This is to be expected since nearly all of the operations performed involving

the individual components of the vectors which are used to represent the variable precision interval data type are performed at this level. This includes the normalization of the results as well as the carrying out of the proper roundings on these results.

In summary, each level of the multilevel interpretive support structure operates in a cooperative manner, with its own clearly defined responsibilities. The top two levels, providing control of the operations upon the interval variables, incurs only a small fraction of the overhead associated with the use of the variable precision interval data type. This overhead is fairly independent of the precision. This independence is due to the fact that the contents of the vectors are almost never operated upon at these levels. Any variance encountered is introduced by an occasional copy or compare operation that must be performed on interval variables at these levels. The majority of overhead is incurred by the third level, which actually performs the operations involving the vectors which represent the variable precision interval data type. The amount of overhead at this level is entirely dependent on the precision, the relationship being exponential in nature rather than linear, Figure 24. This dependence on precision is to be expected as this level deals exclusively with the vectors.

4.2 Summary

Interval arithmetic can, at times, be extremely useful. For instance, it can be used to indicate the limits of precision of an algorithm for a given set of data. From the testing it was shown that much better bounds on the results could be obtained using the variable interval package. This was, of course, not unexpected. The price paid was in runtime efficiency. The use of standard precision intervals resulted in approximately an order of magnitude increase in execution time over that of single or double precision arithmetic. 56 decimal interval arithmetic resulted in a further increase of more than one to more than two orders of magnitude. Variable precision interval arithmetic with precision 56 resulted in an increase of three orders of magnitude. It should be noted here that the 56 digit version was based upon the 59 decimal digit hardware arithmetic unit of the Honeywell H68/80 processor. The software simulated basic operations of the variable precision interval arithmetic caused that arithmetic to take much longer.

One obvious application of variable precision interval arithmetic would be to validate existing programs. Any data sensitivity discovered could be included in a description of the algorithm and directions for its use. Although variable precision interval arithmetic is expensive, its cost must be balanced against possible consequences of using invalid results.

An organization like the Corps of Engineers might weigh the possibility of a defective dam or the cost of moving 100,000 tons of dirt against the cost of a few hours of computer time.

A more effective technique would be to first test the algorithm using single precision interval arithmetic. Its relatively small decrease in run time efficiency indicates that its use is more than justified as an economical means of identifying possible trouble areas in an algorithm for the data under consideration. The more expensive variable precision interval package could be applied to just those cases where possible trouble areas have been identified. Variable precision interval arithmetic can be used to determine the precision of the arithmetic required to guarantee a given significance in the results of an algorithm. Arbitrarily picking a given precision for arithmetic does not guarantee results in which absolute confidence can be placed. How much more confidence can one have in results obtained on a 60 bit word machine than in results obtained on a 36 bit word machine?

In general, whether using interval or regular arithmetic, the greater the precision the longer the run time required for a given algorithm. Having variable precision interval arithmetic would allow the validation of algorithms for which standard precision interval arithmetic is insufficient. Further, the cost of this validation could be held to a minimum by making full use

of the ability to specify different precisions for different variables. Computations with high precision requirements could be performed with the necessary precision while those less computationally demanding could be performed with a lower, less cpu consuming, precision. In any case, the overhead associated with execution in interval arithmetic will only be as great as required for the necessary precision.

The large amount of processor time needed for variable precision interval arithmetic is its major drawback. The execution speed of interval arithmetic can be increased in several ways. One would be to decrease the number of levels of interpretation required in the current implementation. The optimum solution would be to have a hardware or firmware module which could execute variable precision interval arithmetic. Many existing minicomputer systems have undefined opcodes for just such requirements. As a side effect, an arithmetic unit that can execute variable precision interval arithmetic can also execute traditional variable precision floating point arithmetic. This means that interval arithmetic could be used to determine the required arithmetic precision needed to obtain results of the desired accuracy. The algorithm, then, could be executed using only that precision.

[1] Ladner, T. D. and Yohe, J. M., "An interval arithmetic package for the UNIVAC 1108," The University of Wisconsin, Mathematics Research Center, Technical Summary Report No. 1055, May, 1970.

[2] Yohe, J. M., "Best possible floating point arithmetic," The University of Wisconsin, Mathematics Research Center, Technical Summary Report No. 1054, March, 1970.

[3] Binstock, W., Hawkes, J. and Hsu, N., "An interval input/output package for the UNIVAC 1108," The University of Wisconsin, Mathematics Research Center, Technical Summary Report No. 1212, September, 1973.

[4] Crary, F. D., "The AUGMENT precompiler, I. User information," The University of Wisconsin, Mathematics Research Center, Technical Summary Report No. 1469, December, 1974.

[5] Crary, F. D., "The AUGMENT precompiler, II. Technical documentation," The University of Wisconsin, Mathematics Research Center, Technical Summary Report No. 1470, October, 1975.

[6] Moore, R. E, Interval Analysis, Prentice-Hall Inc., Englewood Cliffs, N. J., 1966.

[7] Abramovitz, M. and Stegun, I. A., (ed.), Handbook of Mathematical Functions, National Bureau of Standard Applied Mathematics Series, June, 1964.

[8] Podlaska-Lando, S. and Reuter Eric K., "Implementation and Evaluation of Interval Arithmetic Software, Report 2: The Honeywell Multics System", Technical Report No. 0-79-1, Office, Chief of Engineers, U. S. Army, Washington D. C. April 1979.

[9]. Software Tools, Kernighan, B. and Plauger, P., Addison-Wesley Publishing Company, Reading, Massachusetts, 1976.

[10] Brent, R. P., "A FORTRAN multiple-precision arithmetic package," Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, May, 1976.

[11] Yohe, J. M., "Software for interval arithmetic: a reasonably portable package," Transactions on Mathematical Software, to be published.

[12] Ryder, B. G., "The PFORT Verifier", Software Practice and Experience, Vol. 4, 1974.

REFERENCES

Appendix A.

RATFOR

RATFOR is a preprocessor for FORTRAN (written in FORTRAN) which allows the use of contemporary control structures such as DO-WHILE, IF-THEN-ELSE and REPEAT-UNTIL [9]. RATFOR is unique in that it has the advantages of being highly portable, easily used and fairly efficient. The primary purpose of RATFOR is to make FORTRAN a better programming language by permitting and encouraging the writing of readable and well-structured programs. This is done by providing the control structures that are unavailable in FORTRAN, and by improving the "cosmetics" of the language.

The cosmetic aspects of RATFOR have been designed to make it concise and reasonably pleasing to the eye. It is free-form. That is, statements may appear anywhere on an input line. Other additions also improve the readability of the language. For example, the use of the symbol ">" conveys the meaning of code more rapidly than equivalent strings of symbols such as ".GT.".

To show the advantages of RATFOR consider the construct

IF (condition) THEN (s1) ELSE (s2).

This construct is, for the most part, fairly easy to understand. IF the "condition" is true THEN statements "s1" are to be executed, ELSE statements "s2" are to be executed. However, this

construct is rather awkward to express in FORTRAN. AS an example, suppose that if the value of the variable X were greater than or equal to 10.7, then X is to be divided by 18.3 and the counter KOUNT incremented by one. However, if the condition were false then the variable X is to be multiplied by 18.3 and the counter KOUNT decremented by one. One way of expressing this in FORTRAN as shown in Figure 32.

```
      IF (X.GE.10.7) GO TO 10
      X = X * 18.3
      KOUNT = KOUNT - 1
      GO TO 20
10    X = X / 18.3
      KOUNT = KOUNT + 1
20    CONTINUE
```

FIGURE 32

Example of FORTRAN code

On the other hand, the same logic could be expressed in RATFOR as show in Figure 33.

```
      IF (X >= 10.7)
        {
         X = X / 18.3
         KOUNT = KOUNT + 1
        }
      ELSE
        {
         X = X * 18.3
         KOUNT = KOUNT - 1}
```

FIGURE 33

RATFOR version of FORTRAN code in Figure 32

With these additional contemporary programming language

constructs the programmer is able to produce a readable, structured program and have it translated automatically into standard FORTRAN. It is herein that RATFOR's greatest value lies. A highly readable, structured program is a program that is easier to develop, debug and modify. A programmer assigned to modify an existent program is able to get the job done quickly in direct proportion to the understandability of the code. By the automatic translation of these constructs into FORTRAN, the programmer is able to devote a larger portion of time to the development of ideas rather than their translation while being assured that this translation will be done correctly each and every time.

RATFOR is written in a portable version of FORTRAN. The installation of RATFOR entails the production of an object code deck for RATFOR and the providing for input and output attachments. The input for RATFOR must be attached to the FORTRAN logical unit number 10. The output for RATFOR is written to FORTRAN logical unit number 11. Error conditions are displayed on FORTRAN logical unit number 12. There is also a version of RATFOR written in RATFOR which is a quite useful aid in understanding the translation process used by RATFOR. The RATFOR version also supplies an excellent example of the use of RATFOR. The rest of this appendix provides a summary of the statements and operators accepted by RATFOR.

RATFOR statements:

1. RATFOR STATEMENT -- One or more FORTRAN statements enclosed in brackets. (Brackets optional for a single FORTRAN statement.)

```
EXAMPLE:
        [ X = 1
          READ (5,1) Z]
```

Conditional statements:

2. IF STATEMENTS

A. IF (condition)
RATFOR STATEMENT

```
EXAMPLE:
      IF (X.EQ.Y) [
            OLDX = X
            X = Y**2]
```

B. IF (condition)
RATFOR STATEMENT
ELSE
RATFOR STATEMENT -- ELSE clause is optional

```
EXAMPLE:
      IF (X == Y)
          [
            OLDX = X
            X = Y**2
          ]
      ELSE X = X**2
```

C. IF (condition)
RATFOR STATEMENT
ELSE IF (condition)
RATFOR STATEMENT
ELSE
RATFOR STATEMENT

```
EXAMPLE:
      IF (X.EQ.Y)
          Z = 0
      ELSE if (X > Y)
          [
           OLDX = X
           X = Y**2]
      ELSE
          [Z = 1
           Y = 0
           X = 0]
```

DO_statement:

3. DO index = initial, final, step
RATFOR STATEMENT

EXAMPLE:

```
DO I = 2,100,2
   [X(I) = X(I-1) * X(I)
    X(I-1) = Y(I-1)
   ]
```

Loop_control_statements:

4. BREAK -- exit from loop

5. NEXT -- go to bottom of loop

Looping structures:

6. WHILE (condition)
RATFOR STATEMENT

EXAMPLE:

```
WHILE (X >= LIMIT)
  [ SUM = SUM + X
    X = KURD(X)
  ]
```

7. FOR (initialization; condition; increment)
RATFOR STATEMENT -- increment and initialization are FORTRAN statements; condition is a FORTRAN logical expression.

EXAMPLE:

```
FOR (I = 1; X < Y; I = I + 2)
   [ X = KURD (X,I)
     SUMX = SUMX + X
   ]
```

8. REPEAT
RATFOR STATEMENT
UNTIL (condition)

EXAMPLE:

```
REPEAT
  [SUM = SUM + X
   X = KURD(X)
  ]
UNTIL (X.LT.LIMIT)
```

Miscellaneous:

9. # -- comment statement

EXAMPLE:

```
# this is a comment statement
X = 1 # X is assigned the value of one
```

10. % -- do not process the remainder of the line; just shift left one column. (Used to convert FORTRAN comment statements to RATFOR)

11. DEFINE label value -- value will be substituted for label throughout the program

EXAMPLE:

DEFINE YES 1 (replaces "YES" with "1" throughout program)

12. logical operators -- >=, <=, ==, >, <, !, &, ^
(ge, le, eq, gt, lt, or, and, not)

13. STATEMENT NUMBERS -- if first field in statement is numeric, it is assumed to be a statement number.

14. INCLUDE n -- begin reading input from FORTRAN I/o unit n. This is a very primitive include mechanism.

Appendix B

AUGMENT Interface

AUGMENT is a preprocessor which allows the introduction of non-standard data types (e.g. multiple precision interval numbers) into FORTRAN programs. The introduction of a data type is accomplished by passing the program containing the extended data types to AUGMENT along with an AUGMENT description deck. The description deck contains the necessary information needed by AUGMENT to properly translate the extended data types and the operations performed upon them into standard FORTRAN. This greatly simplifies the task of writing a program for a multiple-precision interval computation, or converting a single (or double) precision routine to multiple precision.

For example, if AUGMENT is used we can write expressions such as shown in Figure 34 where X, Y, and Z are multiple precision. This will automatically be translated to the FORTRAN equivalent as shown in Figure 35.

```
VARINTERVAL X, Y, Z
        .
        .
        .
X = Y + Z*EXP(X+1)/Y
```

FIGURE 34

Portion of Stuctured Variable Precision Interval FORTRAN code

```
      INTEGER X(24), Y(24), Z(44)
                    .
                    .
                    .
      CALL MVADDI (X, 1, MPTEMP)
      CALL MVEXP (MPTEMP, MPTEMP)
      CALL MVMUL (Z, MPTEMP, MPTEMP)
      CALL MVDIV (MPTEMP, Y, MPTEMP)
      CALL MVADD (Y, MPTEMP, X)
```

FIGURE 35

Standard FORTRAN equivalent of Figure 34

The description deck which specifies the variable precision interval package to AUGMENT is shown in Figure 36. The AUGMENT description deck contains 7 major sections. The first section instructs AUGMENT on how the data type is actually to be declared in the FORTRAN output. This is very similar to the PASCAL type declaration. The next section gives details on how operations upon the extended data type are to be translated. For example, if A and B are of type VARINTERVAL then A+B would be translated as CALL MVADD(X,Y,RESULT). The third section is supplied for the extraction of the sign of a variable precision interval variable. The fourth section is supplied so that information concerning the inner components of the structured variable precision data type may be extracted. The following section gives instructions on the conversion of functional references. Its function is basically the same as the second section. The sixth section contains conversion information. For example, conversion from real to variable precision interval would entail a call to MVCRM. The last section indicates which routine is to be called to perform

assignments. In this case it is MVSTR.

```
*DESCRIBE VARINTERVAL
DECLARE INTEGER, KIND SAFE SUBROUTINE, PREFIX MV
OPERATOR + (,NULL UNARY, PRV, $), - (NEG, UNARY),
         + (ADD, BINARY3, PRV, $, $, $, COMM), * (MUL),
         - (SUB,,,,,, NONCOMM), / (DIV), ** (PWR2),
         + (ADDI,,,, INTEGER), * (MULI), / (DIVI), ** (PWR),
         .EQ. (EQ, BINARY2, PRV, $, LOGICAL, COMM),
         .NE. (NE), .GE. (GE,,,,, NONCOMM), .GT. (GT),
         .LE. (LE), .L           T. (LT)
TEST     MPSIGA (SIGA, INTEGER)
FIELD    SGN (SIGA, SIGB, ($), INTEGER),
         EXPON (EXPA, EXPB), BASE (BASA, BASB),
         NUMDIG (DIGA, DIGB), MAXEXP (MEXA, MEXB),
         DIGIT (DGA, DGB, ($, INTEGER))
FUNCTION ABS (ABS, ($), $), ASIN (ASIN), ATAN (ATAN),
         COS (COS), COSH (COSH),EXP (EXP), INT (CMIM),
         LN (LN), LOG (LN), SIN (SIN), SINH (SINH),
         SQRT (SQRT), TAN (TAN), TANH (TANH),
         MAX (MAX, ($, $)), MIN (MIN), ROOT (ROOT),
         MPINF (INF(SUBROUTINE),($,INTEGER,INTEGER,
         HOLLERITH),LOGICAL), MPOUTF (OUTF(SUBROUTINE)),
         MPINF (INF(SUBROUTINE), ($, INTEGER, INTEGER,
         INTEGER)), MPOUTF (OUTF(SUBROUTINE))
CONVERSION CTM (CDM, DOUBLE PRECISION, $, UPWARD),
           CTM (CIM, INTEGER), CTM (CRM, REAL),
           CTD (CMD(SUBROUTINE),$,DOUBLE PRECISION,DOWNWARD),
           CTI (CMI(SUBROUTINE),, INTEGER),
           CTR (CMR(SUBROUTINE),, REAL)
SERVICE COPY (STR)
COMMENT    END OF AUGMENT DESCRIPTION DECK FOR MP PACKAGE
```

FIGURE 36

AUGMENT description deck for the Variable Precision Interval data type

Appendix C

The Basic MP Package

1. General Description of the MP Package

MP is a multiple precision arithmetic package[10]. It is almost completely machine independent, and should run on any machine with an ANSI standard FORTRAN compiler, sufficient memory, and a wordlength of at least 16 bits. The machine dependent sections are those which deal with packed multiple precision numbers. Some modifications would be necessary for a wordlength of less than 16 bits.

MP has been tested on a Univac 1108 (e level FORTRAN v), a Univac 1100/42 (e and T level FORTRAN v, ascii FORTRAN), a PDP 10 (FORTRAN 10 and FORTRAN 40), an IBM 360/50 (FORTRAN g and FORTRAN h, opt = 2), an IBM 360/91 and 370/168 (FORTRAN h extended, opt = 2), a Cyber 76 (ftn 4.2, opt = 1), a PDP 11/45 (dos), and a Honeywell 68/80 (Multics release 6.1). These machines have effective integer word lengths ranging from 16 to 48 bits.

MP numbers are in normalized floating point format as shown in Figure 37. The base (B) and number of digits (T) are arbitrary (subject to some restrictions given below), and may be varied dynamically.

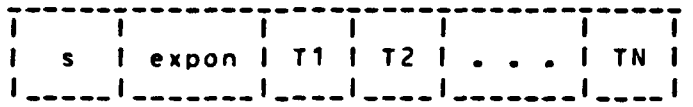


FIGURE 37

Multiple precision number format. s = sign (0, -1 or +1) expon = exponent (to base B) Ti = digit (in base B) Note that words 2 to T+2 are undefined if sign = 0.

Arithmetic is rounded, and four guard digits are used for addition and multiplication, so the correctly rounded result is produced. Division, sqrt etc are done by Newtons method, but give the exact result if it can be represented with T-2 digits. Other routines (mpsin, mpln etc) usually give a result y = f(x) which could be obtained by making an o(B**(1-T)) perturbation in x, evaluating f exactly, then making an o(B**(1-T)) perturbation in y.

Exponents can lie in the range -M, ... , +M inclusive, where M is set by the user. On underflow during an arithmetic operation, the result is set to zero by subroutine MPUNFL. On overflow subroutine MPOVFL is called and execution is terminated with an error message. Error messages are printed on logical unit LUN, where LUN is set by the user, and then execution is terminated by a call to subroutine MPERR. It is assumed that logical records of up to 80 characters may be written on unit LUN. A working array of size MXR (see below) must be provided in common.

The parameters B, T, M, LUN and MXR are passed to the utility routines in common, together with a working array R which must be sufficiently large (see below). Most routines use the statements

```
COMMON B, T, M, LUN, MXR, R
INTEGER B, T, R(1)
```

and it is assumed that R is dimensioned sufficiently large in the calling program, and that MXR is set to the dimension of R in the calling program.

It is assumed that the compiler passes addresses of arrays used as arguments in subroutine calls (i.e., call by reference), and does not check for array bounds violations (either for arguments or for arrays in common). Apart from these violations, MP is written in ANSI standard FORTRAN (ANSI x3.9-1966). This has been checked by the pfort verifier. The only machine-dependent routine is MPUPK (which unpacks characters stored several to a word). Other routines which may require trivial changes are MPSET (which causes an integer overflow), MPINIT and TIMEMP (see comments below).

2. Constraints

There are several constraints that must observed in using the MP package. They are:

1) The base B must be at least 2.

2) T (number of digits) must be at least 2.

3) M (exponent range) must be greater than T and less than 1/4 the largest machine-representable integer.

4) 8*B**2-1 must be no greater than the largest machine- representable integer

5) The integers 0, 1, ... , B must be exactly representable as single precision floating point numbers

6) B**(T-1) should be at least 10**7.

B and T may be set to give the equivalent of a specified number of decimal places by calling MPSET (see below), or may be set directly by the user. If MPSET is not called, the user must remember to initialize M, LUN and MXR (see above) as well as B and T before calling any MP routines. (It would be possible to use labelled common instead of blank common throughout, and set default initializations in a data statement.)

To conserve space choose B fairly large, subject to the natural restrictions of word size and the constraint given above. Maximum values for the base for various word sizes are given below in Figure 38. The figures given as a power of ten are useful in that their use makes for easier debugging of user programs which call the MP package. It is, for example, much easier for the user to determine the base ten value of a digit in base 10000 rather than a digit in base 16384.

48 bits, could use B = 4194304 or 1000000
36 bits, could use B = 65536 or 10000
32 bits, could use B = 16384 or 10000
24 bits, could use B = 1024 or 1000
18 bits, could use B = 128 or 100
16 bits, could use B = 64 or 10

FIGURE 38

Maximum values of base for various word sizes

Avoid multiplication or division by MP numbers, as these take o(T**2) operations, whereas multiplication or division by integers take o(T) operations.

MP numbers used as arguments of subroutines need not be distinct. For example,

CALL MPADD (X, Y, Y) or CALL MPEXP (X, X)

are acceptable. However, distinct arrays which overlap should not be used.

The MP package used with the interval data type extension has been modified to incorporate the proper roundings needed for interval arithmetic. This version of the MP package passes an added parameter to indicate the type of rounding desired. It not only incorporates the directed roundings but the standard rounding and truncation as well.

3. Summary of Available MP Routines

basic arithmetic - mpadd, mpaddi, mpaddq, mpdiv, mpdivi, mpmul, mpmuli, mpmulq, mprec, mpsub

powers and roots - mppwr, mppwr2, mpqpwr, mproot, mpsqrt

elementary functions - mpasin, mpatan, mpcos, mpcosh, mpexp, mpln, mplngs, mplni, mpsin, mpsinh, mptan, mptanh

constants - mpeps, mpmaxr, mpminr, mppi, mppigl

input and output - mpdump, mpin, mpine, mpinf, mpout, mpoute, mpoutf, mpout2

conversion - mpcam, mpcdm, mpcim, mpcmd, cpcmde, mpcmef, mpcmi, mpcmim, mpcmr, mpcmre, mpcqm, mpcrm

comparison - mpcmpa, mpcmpi, mpcmpr, mpcomp, mpeq, mpge, mpgt, mple, mplt, mpne

general utility routines - mpabs, mpclr, mpcmf, mpgcda, mpgcdb, mpinit, mpkstr, mpmax, mpmin, mpneg, mppack, mppoly, MPSET, mpstr, mpunpk

error detection and handling - mpchk, mperr, mpovfl, mpunfl

AUGMENT interface routines - mpbasa, mpbasb, mpdga, mpdgb, mpdiga, mpdigb, mpexpa, mpexpb, mpmexa, mpmexb, mpsiga, mpsigb

miscellaneous routines used by the above - mpadd2, mpadd3, mpart1, mpbes2, mperf2, mperf3, mpexp1, mpext, mpgcd, mphank, mpio, mplns, mpl235, mpmlp, mpmul2, mpnzr, mpsin1, mpupk, mp40d, mp40e, mp40f, mp40g, timemp

4. Restricted Names

When writing programs which use MP via the RATFOR/AUGMENT interface, it is safest to avoid using the following identifiers except with their reserved meaning.

base	see description of mpbasa and mpbasb in section 6.
ctd	see description of mpcmd.
cti	see description of mpcmi.
ctm	see description of mpcam, mpcdm, mpcim, mpcqm,

mpcrm and mpurpk.

ctp	see description of mppack.
ctr	see description of mpcmr.
digit	see description of mpdga and mpdgb.
expon	see description of mpexpa and mpexpb.
frac	see description of mpcmf.
gcd	see description of mpgcda.
initialize	see description of mpinit.
int	see description of mpcmim.
log	see description of mpln and mplni.
maxexp	see description of mpmexa and mpmexb.
mpxxxx	(for any letters or digits xxxx).
multipak	see comments in description deck above.
multiple	see comments in description deck above.
numdig	see description of mpdiga and mpdigb.
sgn	see description of mpsiga and mpsigb.

for the following, if the reserved word is xxxx, see the description of mpxxxx in section 6.

abs, addq, art1, asin, atan, bern, besj, cam, cmf, cmim, cmpa, comp, cos, cosh, cqm, daw, ei, erf, erfc, exp, expl, gam, gamq, li, ln, lngm, lngs, lni, lns, max, min, mulq, qpwr, rec, root, sin, sinh, sqrt, str, tan, tanh, zeta.

Appendix D

Mathematical Basis for Interval Arithmetic

The details of the mathematical basis for interval arithmetic are developed in Moore [6]. The set of interval numbers is the set of all closed intervals on the real number line. An interval may be represented by an ordered pair of real numbers $[a,b]$ where $a \leq b$. If $a = b$, then the interval is said to be degenerate.

The operations of addition, subtraction, multiplication, and division between two intervals (except for the division of one interval by an interval containing zero) are defined as follows where $ is one of the above operations:

$$[a,b] \$ [c,d] = \{x \$ y : x \in [a,b] \text{ and } y \in [c,d]\}$$

Each of the operations of addition, subtraction, multiplication, and division may be defined as follows:

$$[a,b] + [c,d] = [a+c, b+d]$$

$$[a,b] - [c,d] = [a-d, b-c]$$

$$[a,b] * [c,d] = [\min\{ac,ad,bc,bd\}, \max\{ac,ad,bc,bd\}]$$

$$[a,b] / [c,d] = [\min\{a/c, a/d, b/c, b/d\}, \max\{a/c, a/d, b/c, b/d\}] \quad \text{if } 0 \notin [c,d]$$

In the cases of multiplication and division, by examining the signs of the endpoints of the intervals being multiplied or divided; a determination in advance can be made of which products or quotients will be the maximum and the minimum.

The following real single valued functions of intervals may be useful:

The midpoint of an interval, mid ([a,b]), is defined to be the real number (a+b)/2.

The length of an interval, length ([a,b]), is defined to be the real number b-a.

The supremum of an interval, sup ([a,b]), is the real number a.

The infimum of an interval, inf ([a,b]), is the real number b.

The distance from interval [a,b] to interval [c,d], dis ([a,b],[c,d]), is defined to be the real number max{|c-a|,|d-b|}.

The following interval single valued functions of intervals may also be useful:

The union of intervals [a,b] and [c,d], union ([a,b],[c,d]), is defined to be the smallest interval containing both [a,b] and [c,d] and is given by [min{a,c},max{b,d}]. The intersection of intervals [a,b] and [c,d], intsct ([a,b],[c,d]), is defined to be the largest interval contained in each of [a,b] and [c,d] or is empty if [a,b] and [c,d] are disjoint intervals and is given by [max{a,c},min{c,d}].

The relational operations may be defined on intervals as follows:

[a,b] = [c,d] if and only if a = b = c = d

The above definition means that two intervals are equal if and only if they both are degenerate and represent the same real number. This definition is employed instead of the more general definition of testing for a = c and b = d. The reason the more general definition is not used is because we will regard intervals as bounds on an exact but unknown real number. If two intervals were not degenerate and if both intervals had the same endpoints, then the intervals may not represent the same exact real number. The only way for the two intervals to represent the same exact real

number is for both intervals to be degenerate with their endpoints equal to the real number. We also say that

$[a,b] \neq [c,d]$ if and only if $[a,b] \cap [c,d] = \emptyset$

This definition means that two intervals are not equal if and only if they are disjoint intervals and cannot represent the same exact real number.

$[a,b] \leq [c,d]$ if and only if $b \leq c$

The above definition means that two intervals are ordered by the $\leq$ relational operator if and only if $\forall x \in [a,b]$ and $\forall y \in [c,d]$, $x \leq y$.

$[a,b] > [c,d]$ if and only if $a > d$

The above definition means that two intervals are ordered by the $>$ relational operator if and only if $\forall x \in [a,b]$ and $\forall y \in [c,d]$, $x > y$.

Interval valued functions of interval variables are defined in terms of real valued functions of real variables. If f is a real valued function of a real variable, then f may be extended to an interval valued function, F, of an interval variable by defining

$$F([a,b]) = \{f(x) : x \in [a,b]\}$$

If f is defined and continuous on $[a,b]$, then $F([a,b])$ will be an interval. If intervals are to represented as pairs of real numbers, then the above definition is not operational. Some means is needed for deriving the endpoints of the image of $[a,b]$ under the function F. The endpoints of the image interval will be the image under f of points of $[a,b]$.

For functions, f, that are monotonic on the interval [a,b], the endpoints of the image of [a,b] under F can be expressed as the result of the function f evaluated at the endpoints of [a,b]. If f is monotonic increasing on [a,b], then F([a,b]) = [f(a), f(b)]. If f is monotonic decreasing on [a,b], then F([a,b]) = [f(b), f(a)]. If f is not monotonic over [a,b], then [a,b] can be divided into disjoint subintervals; X(i), i = 1,2,3,...n; where U X(i) = [a,b] and f is monotonic on each X(i). In this case F([a,b]) = U f(X(i)).

Appendix E

Description of MV Routines Available

1. Description of Available Variable Precision Interval Routines

The suggested method of calling the MV Routine directly is given first. Second (third, ...) alternative method(s) (if any) may be used when the AUGMENT interface, described earlier, is used to process the user program. Unless otherwise noted, X, Y, Z represent MV numbers I, J, K, L, IX etc. represent integers, RX, RY etc. represent reals, and DX, DY etc. denote double precision numbers. See Appendix C for definitions of B, T, M, LUN, MXR, R etc. Space required means the dimension of R in common. If not specified, space required is no more than 2*T+4 words. if not specified, space required is no more than 2*T+4 words.

MVABS ******

usage -- CALL MVABS (X, Y) or Y = ABS (X)

description -- sets Y = ABS(X) for MV numbers X and Y

MVADD ******

usage -- CALL MVADD (X, Y, Z) or Z = X + Y

description -- adds X and Y, forming result in Z, where X, Y and Z are MV numbers. four guard digits are used, and then R*-rounding.

MVASIN ******

usage -- CALL MVASIN (X, Y) or Y = ASIN (X)

description -- returns Y = ARCSIN(X), assuming ABS(X) .le. 1, for MV numbers X and Y. Y is in the

range -pi/2 to +pi/2. method is to use MVATAN, so time is o(M(T)T). dimension of R must be at least 5T+12

MVATAN ******

usage -- CALL MVATAN (X, Y) or Y = ATAN (X)

description -- returns Y = ARCTAN(X) for MV X and Y, using an o(T.M(T)) method which could easily be modified to an o(SQRT(T)M(T)) method (as in MPEXP1). Y is in the range -pi/2 to +pi/2. for an asymptotically faster method, see - fast multiple- precision evaluation of elementary functions (by R. P. Brent), J. ACM 23 (1976), 242-251, and the comments in MPPIGL. dimension of R in calling program must be at least 5T+12

MVCDM ******

usage -- CALL MVCDM (DX, Z) or Z = DX

description -- converts double precision number DX to multiple-precision Z. some numbers will not convert exactly on machines with base other than two, four or sixteen. this routine is not called by any other routine in MV, so may be omitted if double precision is not available.

MVCIM ******

usage -- CALL MVCIM (IX, Z) or Z = IX

description -- converts integer IX to multiple-precision Z. note - IX should not be the same location as Z(1) in CALL.

MVCLR ******

usage -- CALL MVCLR (X, N)

description -- sets X(T+3), ... , X(N+2) to zero, useful if precision is going to be increased.

MVCMD ******

usage -- CALL MVCMD (X, DZ) or DZ = X

description -- converts multiple-precision X to double precision DZ. assumes X is in allowable range for double precision numbers. there

is some loss of accuracy if the exponent is large.

MVCMI ******

usage -- CALL MVCMI (X, IZ) or IZ = X

description -- converts multiple-precision X to integer IZ, assuming that X not too large X is truncated towards zero. if int(X)is too large to be represented as a single precision integer, IZ is returned as zero. the user may check for this possibility by testing if ((X(1).ne.0).and.(X(2).gt.0).and.(IZ.eq.0)) is true on return from MVCMI.

MVCMR ******

usage -- CALL MVCMR (X, RZ) or RZ = X

description -- converts multiple-precision X to single precision RZ. assumes X in allowable range. there is some loss of accuracy if the exponent is large.

MVCOMP ******

usage -- J = MVCOMP (X, Y)

description -- compares the multiple-precision numbers X and Y, returning +1 if X .gt. Y, -1 if X .lt. Y, and 0 if X .eq. Y.

MVCOS ******

usage -- CALL MVCOS (X, Y) or Y = COS (X)

description -- returns Y = COS(X) for MV X and Y, using MVSIN dimension of R in common at least 5T+12.

MVCOSH ******

usage -- CALL MVCOSH (X, Y) or Y = COSH (X)

description -- returns Y = COSH(X) for MV numbers X and Y, X not too large. uses MVEXP, dimension of R in common at least 5T+12

MVCRM ******

usage -- CALL MVCRM (RX, Z) or Z = RX

description -- converts single precision number RX to multiple-precision Z. some numbers will not convert exactly on machines with base other than two, four or sixteen.

MVDIV ******

usage -- CALL MVDIV (X, Y, Z) or Z = X/Y

description -- sets Z = X/Y, for MV X, Y and Z. MPERR is called if Y is zero. dimension of R in calling program must be at least 4T+10 (but Z(1) may be R(3T+9)).

MVEQ ******

usage -- if (MPEQ (X, Y)) ... or if (X .EQ. Y) ...

description -- returns logical value of (X .EQ. Y) for MV X and Y. MVEQ Must be declared logical unless augment interface is used.

MVEXP ******

usage -- CALL MVEXP (X, Y) or Y = EXP (X)

description -- returns Y = EXP(X) for MV X and Y. EXP of integer and fractional parts of X are computed separately. see also comments in MPEXP1. time is o(SQRT(T)M(T)). dimension of R must be at least 4T+10 in calling program

MVGE ******

usage -- if (MPGE (X, Y)) ... or if (X .GE. Y) ...

description -- returns logical value of (X .GE. Y) for MV X and Y. MVGE Must be declared logical unless augment interface is used.

MVGT ******

usage -- if (MPGT (X, Y)) ... or if (X .GT. Y) ...

description -- returns logical value of (X .GT. Y) for MV X and Y. MVGT Must be declared logical unless augment interface is used.

MVGET ******

usage -- CALL MVGET (X)

description -- converts the fixed-point decimal number (read under na1 format) in c(1) ... c(n) to a

multiple-precision number in X. Warnings are given for invalid intervals and when the number of digits in the input exceeds the precision of the target.

MVINIT ******

usage -- CALL MVINIT (I) or INITIALIZE MV

description -- declares blank common (used by MV package) and calls MVSET to initialize parameters. I is a dummy integer argument. the augment declaration initialize MV causes a CALL to MVINIT to be generated.

MVLN ******

usage -- CALL MVLN (X, Y) or Y = LN (X) or Y = LOG (X)

description -- returns Y = LN(X), for MV X and Y, using MVLNS. restriction - integer part of LN(X) must be representable as a single precision integer. time is o(SQRT(T).M(T)). dimension of R in calling program must be at least 6t+14.

MVLT ******

usage -- if (MPLT (X, Y)) ... or if (X .LT. Y) ...

description -- returns logical value of (X .LT. Y) for MV X and Y. MVLT Must be declared type logical unless augment interface used.

MVMUL ******

usage -- CALL MVMUL (X, Y, Z) or Z = X*Y

description -- multiplies X and Y, returning result in Z, for MV X, Y and Z.

MVNE ******

usage -- if (MPNE (X, Y)) ... or if (X .NE. Y) ...

description -- returns logical value of (X .NE. Y) for MV X and Y. MVNE Must be declared type logical unless augment interface used.

MVNEG ******

usage -- CALL MVNEG (X, Y) or Y = -X

description -- sets Y = -X for MV numbers X and Y

MVPUT ******

usage -- CALL MVPUT (X, W, N, LUN)

description -- converts multiple-precision X to FW.N format and outputs the result to the FORTRAN logical unit specified by LUN.

MVOVFL ******

usage -- CALL MVOVFL (X)

description -- called on multiple-precision overflow, ie when the exponent of MV number X would exceed M. at present execution is terminated with an error message after calling MPMAXR(X), but it would be possible to return, possibly updating a counter and terminating execution after a preset number of overflows. Action could easily be determined by a flag in labelled common.

MVPI ******

usage -- CALL MVPI (X)

description -- sets MV X = pi to the available precision. uses pi/4 = 4.arctan(1/5) - arctan(1/239).

MVPWR ******

usage -- CALL MVPWR (X, N, Y) or Y = X**N

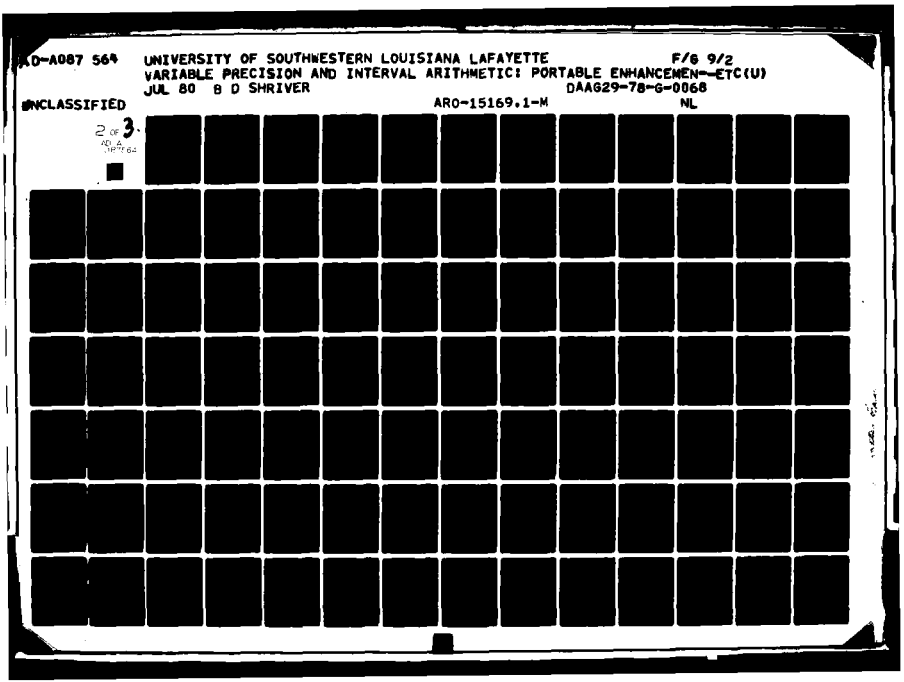
description -- returns Y = X**N, for MV X and Y, integer N, with 0**0 = 1. R must be dimensioned at least 4T+10 in calling program (2t+6 is enough if N nonnegative).

MVROOT ******

usage -- CALL MVROOT (X, N, Y) or Y = root (X, N)

description -- returns Y = X**(1/N) for integer n, ABS(N) .LE. max (B, 64). and MV numbers X and Y, using Newtons method without divisions. space = 4T+10 (but Y(1) may be R(3T+9))

AD-A087 564 UNIVERSITY OF SOUTHWESTERN LOUISIANA LAFAYETTE F/G 9/2
VARIABLE PRECISION AND INTERVAL ARITHMETIC: PORTABLE ENHANCEMEN--ETC(U)
JUL 80 B D SHRIVER DAAG29-78-G-0068
UNCLASSIFIED ARO-15169.1-M NL



MVSET ******

usage -- CALL MVSET (lunit, idecpl, itmax2, maxdr)

description -- sets base (B) and number of digits (T) to give the equivalent of at least idecpl decimal digits. idecpl should be positive. itmax2 is the dimension of arrays used for MV numbers, so an error occurs if the computed T exceeds itmax2 - 2. MVSET also sets LUN = lunit (logical unit for error messages), MXR = maxdr (dimension of R in common, .ge. T+4), and M = (w-1)/4 (maximum allowable exponent), where w is the largest integer of the form 2**K-1 which is representable in the machine, K .le. 47 (on most machines K = one less than number of bits per word, but this is not true on cdc 6000/7000 machines). The computed B and T satisfy the conditions (T-1)*ln(B)/ln(10) .ge. idecpl and 8*B*B-1 .le. w . approximately minimal T and maximal B satisfying these conditions are chosen. parameters lunit, idecpl, itmax2 and maxdr are integers. beware - MVSET will cause an integer overflow to occur ****** if wordlength is less than 48 bits. if this is not allowable, change the determination of w (do 30 ... to 30 w = wn) or set B, T, M, LUN and MXR without calling MVSET.

MVSIN ******

usage -- CALL MVSIN (X, Y) or Y = SIN (X)

description -- returns Y = SIN(X) for MV X and Y, dimension of R in calling program must be at least 5T+12

MVSINH ******

usage -- CALL MVSINH (X, Y) or Y = SINH (X)

description -- returns Y = SINH(X) for MV numbers X and Y, X not too large.

MVSQRT ******

usage -- CALL MVSQRT (X, Y) or Y = SQRT (X)

description -- returns Y = SQRT(X), using subroutine MVROOT if X .GT. 0. dimension of R in calling program must be at least 4T+10 (but

Y(1) may be R(3T+9)). X and Y are MV numbers.

MVSTR ******

usage -- CALL MVSTR (X, Y) or Y = X

description -- sets Y = X for MV X and Y.

MVSUB ******

usage -- CALL MVSUB (X, Y, Z) or Z = X - Y

description -- subtracts Y from X, forming result in Z, for MV X, Y and Z.

MVTAN ******

usage -- CALL MVTAN (X, Y) or Y = TAN (X)

description -- sets Y = TAN(X) for MV X and Y

MVTANH ******

usage -- CALL MVTANH (X, Y) or Y = TANH (X)

description -- returns Y = TANH(X) for MV numbers X and Y,

MVUNFL ******

usage -- CALL MVUNFL (X)

description -- called on multiple-precision underflow, ie when the exponent of MV number X would be less than -M. the underflowing number is set to zero. an alternative would be to CALL MVMINR (X) and/or return, possibly updating a counter and terminating execution after a preset number of underflows. action could easily be determined by a flag in labelled common.

2. Summary of Available Variable Precision Interval Routines

Basic Arithmetic - MVADD, MVDIV, MVMUL, MVSUB

Powers and Roots - MVPWR, MVROOT, MVSQRT

Elementary Functions - MVASIN, MVATAN, MVCOS, MVCOSH, MVEXP, MVLN, MVSIN, MVSINH, MVTAN, MVTANH

Constants - MVPI

Input and Output - MVPUT, MVGET

Conversion - MVCDM, MVCIM, MVCMD, MVCMI, MVCMR, MVCRM

Comparison - MVCOMP, MVEQ, MVGE, MVGT, MVLE, MVLT, MVNE

General Utility Routines - MVABS, MVCLR, MVINIT, MVNEG, MVSET, MVSTR

AUGMENT Interface Routines - MVBASA, MVBASB, MVDGA, MVDGB, MVDIGA, MVDIGB, MVEXPA, MVEXPB, MVMEXA, MVMEXB, MVSIGA, MVSIGB

Appendix F.

Sample Program Source and FORTRAN Output

1. Program to Compute the Constant e

1.1 VARINTERVAL Source

```
# THIS IS A PROGRAM TO COMPUTE e
      DATA SUM(1), NFACT(1), STEP(1),
                   ERROR(1) /20,30,40,50/
      VARINTERVAL SUM, NFACT, STEP, ERROR

# INITIALIZATION
      SUM = 0
      NFACT = 1
      STEP = 1
      ERROR = 1.0E-6
      I = 0
# INITIALIZE THE CPU AND PAGING COUNTERS
      CALL CTP (1)

# LOOP THROUGH UNTIL STEP IS LESS THAN OR EQUAL TO ERROR
      REPEAT
        [ SUM = SUM + STEP
          I = I + 1
          NFACT = NFACT * FLOAT(I)
          STEP = 1/NFACT ]
      UNTIL (STEP <= ERROR)

# DISPLAY THE RESULTS
      WRITE(6,2)
2     FORMAT (1X,"e IS EQUAL TO")
      CALL MVPUT (SUM)
      WRITE(6,1) I
1     FORMAT (1h+,T7,"COMPUTED IN ",I3," STEPS")

# PRESENT THE CPU TIME AND PAGING FOR THIS PROCEDURE
      CALL CTP(0)
      STOP
      END
```

1.2 FORTRAN_Output

```
C               ***** PROCESSED BY AUGMENT VERSION ,4J *****
C                 ----- TEMPORARY STORAGE LOCATIONS -----
C                       VARINTERVAL
      INTEGER MVTMP(104,1)
C                 ----- LOCAL VARIABLES -----
      INTEGER I
C                       VARINTERVAL
      INTEGER ERROR(104), NFACT(104), STEP(104), SUM(104)
C                 ----- GLOBAL VARIABLES -----
      INTEGER MVBASE, MVDIS, MVLUN, MVM, MVMAXT, MVMXR,
     *     MVR(1040), MVT, MVTEMP(104)
C                 ----- SUPPORTING PACKAGE FUNCTIONS -----
      LOGICAL MVLE
C                 ----- COMMON BLOCKS -----
      COMMON // MVDIS, MVBASE, MVT, MVM, MVLUN, MVMXR, MVR
      COMMON /MVTEMP/ MVMAXT, MVTEMP
C                 ***** TRANSLATED PROGRAM *****
C ***** UNRECOGNIZED STATEMENT *****
       DATA SUM(1), NFACT(1), STEP(1),
     *                   ERROR(1) /20,30,40,50/
C                 ----- BEGIN INITIALIZATION -----
      CALL MVINIT (MVTMP(1,1),1)
      CALL MVINIT (ERROR,0)
      CALL MVINIT (NFACT,0)
      CALL MVINIT (STEP,0)
      CALL MVINIT (SUM,0)
C                 ----- END INITIALIZATION -----
C
C       MVMAXT = 50
        MVMAXT = 50
C
C       MVLUN = 6
        MVLUN = 6
C
C       MVBASE = 10
        MVBASE = 10
C
C       MVM = 1000
        MVM = 1000
C
C       MVT = 0
        MVT = 0
C
C       MVMXR = 1040
        MVMXR = 1040
C
C       MVDIS = 2
        MVDIS = 2
C
C       MVTEMP(1) = 0
        MVTEMP(1) = 0
```

```
C
C       MVR(1) = 0
        MVR(1) = 0
C
C       SUM = 0
      CALL MVCIM (0,SUM)
C
C       NFACT = 1
      CALL MVCIM (1,NFACT)
C
C       STEP = 1
      CALL MVCIM (1,STEP)
C
C       ERROR = 1.0E-6
      CALL MVCRM (1.0E-6,ERROR)
C
C       I = 0
        I = 0
C
C       CALL CTP (1)
        CALL CTP (1)
C
C       CONTINUE
        CONTINUE
C
C3001   CONTINUE
23001   CONTINUE
C
C       SUM = SUM + STEP
      CALL MVADD (SUM,STEP,SUM)
C
C       I = I + 1
        I = I + 1
C
C       NFACT = NFACT * FLOAT(I)
C ***** MIXED MODE OPERANDS ACCEPTED *****
      CALL MVCRM (FLOAT (I),MVTMP(1,1))
      CALL MVMUL (NFACT,MVTMP(1,1),NFACT)
C
C       STEP = 1/NFACT
C ***** MIXED MODE OPERANDS ACCEPTED *****
      CALL MVCIM (1,MVTMP(1,1))
      CALL MVDIV (MVTMP(1,1),NFACT,STEP)
C
C3002   IF(.NOT.(STEP .LE. ERROR)) GOTO 23001
23002 IF (.NOT.MVLE (STEP,ERROR))  GOTO 23001
C
C3003   CONTINUE
23003   CONTINUE
C
C99      WRITE(6,2)
999      WRITE(6,2)
2        FORMAT (1X,13He IS EQUAL TO)
```

```
C
C       CALL MVPUT (SUM)
        CALL MVPUT (SUM)
C
C       WRITE(6,1) I
        WRITE(6,1) I
1        FORMAT (1h+,T7,12HCOMPUTED IN ,I3,6H STEPS)
C
C       CALL CTP(0)
        CALL CTP(0)
C
C       STOP
        STOP
        END
```

2. Heat_transfer_program

2.1 VARINTERVAL_Source

```
# PROGRAM TO COMPUTE HEAT TRANSFER IN A TEN FOOT IRON BAR
#  HEAT SOURCE IS 500 DEGREES, AMBIENT TEMPERATURE IS 70
#  DEGREES. OUTPUT IS TEMPERATURE OF BAR AT ONE FOOT
#  INTERVALS. INPUT IS TIME AFTER CONTACT WITH HEAT SOURCE
#  IN MINUTES.
#
# INTEGER VARIABLES
     INTEGER FEET, KLENGTH, ITEMP, TIME
#
# VARINTERVAL VARIABLES

     # CONSTANT VALUED VARIABLES
     DATA ERROR(1), PI(1), PISQR(1), L(1), RO(1), C(1),
                    K(1) /7*56/
     DATA  TO(1), T1(1) /2*56/

     # VARIABLE VALUED VARIABLES
     DATA COUNT(1), THETA(1), SUM(1), XSUM(1), X(1) /5*56/
     DATA T(1) /56/

     # VARINTERVAL DECLARATION
     VARINTERVAL PI, PISQR, L, RO, C, K, TO, T1, COUNT,
                 THETA, SUM, XSUM, X, T, ERROR
#
#
# BEGIN PROGRAM


# INITIALIZATIONS
     CALL CTP(1)     # INITIALIZE THE CPU AND PAGING COUNTERS
     PI = 3.1415927
     PISQR = PI * PI
     L = 10 # LENGTH OF BAR
     K = 30
     RO = 7.1 * 62.3
     C = .12
     TO = 70         # AMBIENT TEMPERATURE
     T1 = 500        # TEMPERATURE OF HEAT SOURCE
     ERROR = 1
     DO I = 1, 10  # SET ERROR = 1.E-56
       [ ERROR = ERROR/100000 ]

# GET THE TIME, IN MINUTES
     WRITE (6,1)
1    FORMAT (" ENTER THE TIME IN MINUTES")
     CALL MVGET (THETA)
#
```

```
# CALCULATE TEMPERATURE FOR EACH FOOT
#
      KLENGTH = L
      DO  FEET = 1, KLENGTH
          [
            # COMPUTE AN INITIAL VALUE FOR T
            T = (K*THETA)/(L**2 * RO * C)

            # COMPUTE DISTANCE TO HEAT SOURCE
            X = FEET/L

            # LOOP FOR THE COMPUTATION OF SUM
            SUM = 0
            XSUM = 0  # INITIAL LOOP VALUES
            COUNT = 0
            TIME = 0

             REPEAT
               [ COUNT = COUNT + 1
                 SUM = SUM + XSUM
                 ITEMP = COUNT
                 XSUM =  ((-1)**ITEMP/COUNT)     *
                      (EXP( -(COUNT**2) * (PISQR) * T) *
                      SIN (COUNT*PI*X))
                 IF (ABS(XSUM) < ERROR )
                   TIME = TIME + 1
                 ELSE
                   TIME = 0
                ]
              UNTIL (TIME == 2)

             # COMPUTE THE TEMPERATURE FOR THIS DISTANCE
              T = T0 + (T1 - T0) * (X + (2.0/PI) * SUM)

             # OUTPUT THE FINAL ANSWER FOR THIS FOOT
                WRITE (6,4) FEET
                CALL MVPUT (T)
           4    FORMAT (1X, "THE ANSWER FOR ",I2," FEET IS ")
                WRITE (6,5)
           5    FORMAT (" The number of steps is: ")
                CALL MVPUT (COUNT)
          ]
        CALL CTP(0)
        STOP
        END
```

2.2 Fortran_Output

```
C                ***** PROCESSED BY AUGMENT VERSION ,4J *****
C                ----- INITIALIZE/ERASE INDEXES -----
      INTEGER 00I1
C                ----- TEMPORARY STORAGE LOCATIONS -----
      INTEGER 00ITMP(1)
C                       VARINTERVAL
      INTEGER MVTMP(116,3)
C                ----- LOCAL VARIABLES -----
      INTEGER FEET, I, ITEMP, KLENGTH, TIME
C                       VARINTERVAL
      INTEGER  C(116),   COUNT(116),   ERROR(116),   K(116),
     L(116),
     *     PI(116), PISQR(116), RO(116), SUM(116), T(116),
     *       TO(116), T1(116), THETA(116), X(116), XSUM(116)
C                ----- GLOBAL VARIABLES -----
      INTEGER MVBASE, MVDIS, MVLUN, MVM, MVMAXT, MVMXR,
     *     MVR(1160), MVT, MVTEMP(116)
C                ----- SUPPORTING PACKAGE FUNCTIONS -----
      LOGICAL MVLT
C                ----- COMMON BLOCKS -----
      COMMON // MVDIS, MVBASE, MVT, MVM, MVLUN, MVMXR, MVR
      COMMON /MVTEMP/ MVMAXT, MVTEMP
C                ***** TRANSLATED PROGRAM *****
C ***** UNRECOGNIZED STATEMENT *****
       DATA ERROR(1), PI(1), PISQR(1), L(1), RO(1), C(1),
     *  K(1) /7*56/
C ***** UNRECOGNIZED STATEMENT *****
       DATA  TO(1), T1(1) /2*56/
C ***** UNRECOGNIZED STATEMENT *****
       DATA COUNT(1), THETA(1), SUM(1), XSUM(1), X(1) /5*56/
C ***** UNRECOGNIZED STATEMENT *****
       DATA T(1) /56/
C                ----- BEGIN INITIALIZATION -----
       MVTEMP(1) = 0
       MVR(1) = 0
      DO 30000  00I1 = 1, 3
30000 CALL MVINIT (MVTMP(1,00I1),1)
      CALL MVINIT (C,0)
      CALL MVINIT (COUNT,0)
      CALL MVINIT (ERROR,0)
      CALL MVINIT (K,0)
      CALL MVINIT (L,0)
      CALL MVINIT (PI,0)
      CALL MVINIT (PISQR,0)
      CALL MVINIT (RO,0)
      CALL MVINIT (SUM,0)
      CALL MVINIT (T,0)
      CALL MVINIT (TO,0)
      CALL MVINIT (T1,0)
      CALL MVINIT (THETA,0)
      CALL MVINIT (X,0)
```

```
      CALL MVINIT (XSUM,0)
C                 ----- END INITIALIZATION -----
C
C      MVMAXT = 56
       MVMAXT = 56
C
C      MVLUN = 6
       MVLUN = 6
C
C      MVBASE = 10
       MVBASE = 10
C
C      MVM = 1003
       MVM = 1003
C
C      MVT = 0
       MVT = 0
C
C      MVMXR = 1160
       MVMXR = 1160
C
C      MVDIS = 2
       MVDIS = 2
C
C      MVTEMP(1) = 0
       MVTEMP(1) = 0
C
C      MVR(1) = 0
       MVR(1) = 0
C
C      CALL CTP(1)
       CALL CTP(1)
C
C      PI = 3.1415927
      CALL MVCRM (3.1415927,PI)
C
C      PISQR = PI * PI
      CALL MVMUL (PI,PI,PISQR)
C
C      L = 10
      CALL MVCIM (10,L)
C
C      K = 30
      CALL MVCIM (30,K)
C
C      RO = 7.1 * 62.3
      CALL MVCRM (7.1*62.3,RO)
C
C      C = .12
      CALL MVCRM (.12,C)
C
C      TO = 70
      CALL MVCIM (70,TO)
```

```
C
C       T1 = 500
        CALL MVCIM (500,T1)
C
C       ERROR = 1
        CALL MVCIM (1,ERROR)
C
C       DO 23001 I = 1, 10
        DO 23001 I = 1, 10
C
C       ERROR = ERROR/100000
        CALL MVDIVI (ERROR,100000,ERROR)
C
C3001   CONTINUE
23001   CONTINUE
C
C3002   CONTINUE
23002   CONTINUE
C
C       WRITE (6,1)
        WRITE (6,1)
1        FORMAT (26H ENTER THE TIME IN MINUTES)
C
C       CALL MVGET (THETA)
        CALL MVGET (THETA)
C
C       KLENGTH = L
        CALL MVCMI (L,00ITMP(1))
        KLENGTH=00ITMP(1)
C
C       DO 23003  FEET = 1, KLENGTH
        DO 23003 FEET = 1, KLENGTH
C
C       T = (K*THETA)/(L**2 * RO * C)
        CALL MVMUL (K,THETA,MVTMP(1,1))
        CALL MVPWR (L,2,MVTMP(1,2))
        CALL MVMUL (MVTMP(1,2),RO,MVTMP(1,2))
        CALL MVMUL (MVTMP(1,2),C,MVTMP(1,2))
        CALL MVDIV (MVTMP(1,1),MVTMP(1,2),T)
C
C       X = FEET/L
C =====  MIXED MODE OPERANDS ACCEPTED =====
        CALL MVCIM (FEET,MVTMP(1,1))
        CALL MVDIV (MVTMP(1,1),L,X)
C
C       SUM = 0
        CALL MVCIM (0,SUM)
C
C       XSUM = 0
        CALL MVCIM (0,XSUM)
C
C       COUNT = 0
        CALL MVCIM (0,COUNT)
```

```
C
C       TIME = 0
        TIME = 0
C
C       CONTINUE
        CONTINUE
C
C3005   CONTINUE
23005   CONTINUE
C
C       COUNT = COUNT + 1
      CALL MVADDI (COUNT,1,COUNT)
C
C       SUM = SUM + XSUM
      CALL MVADD (SUM,XSUM,SUM)
C
C       ITEMP = COUNT
      CALL MVCMI (COUNT,00ITMP(1))
      ITEMP=00ITMP(1)
C
C       XSUM =  ((-1)**ITEMP/COUNT)     *
C       (EXP( -(COUNT**2) * (PISQR) * T) *
C       SIN(COUNT*PI*X))
C ===== MIXED MODE OPERANDS ACCEPTED =====
      CALL MVCIM ((-1)**ITEMP,MVTMP(1,1))
      CALL MVDIV (MVTMP(1,1),COUNT,MVTMP(1,1))
      CALL MVPWR (COUNT,2,MVTMP(1,2))
      CALL MVMUL (MVTMP(1,2),PISQR,MVTMP(1,2))
      CALL MVMUL (MVTMP(1,2),T,MVTMP(1,2))
      CALL MVNEG (MVTMP(1,2),MVTMP(1,2))
      CALL MVEXP (MVTMP(1,2),MVTMP(1,2))
      CALL MVMUL (COUNT,PI,MVTMP(1,3))
      CALL MVMUL (MVTMP(1,3),X,MVTMP(1,3))
      CALL MVSIN (MVTMP(1,3),MVTMP(1,3))
      CALL MVMUL (MVTMP(1,2),MVTMP(1,3),MVTMP(1,3))
      CALL MVMUL (MVTMP(1,1),MVTMP(1,3),XSUM)
C
C       IF(.NOT.(ABS(XSUM) .LT. ERROR )) GOTO 23008
      CALL MVABS (XSUM,MVTMP(1,1))
      IF (.NOT.MVLT (MVTMP(1,1),ERROR))  GOTO 23008
C
C       TIME = TIME + 1
        TIME = TIME + 1
C
C       GOTO 23009
        GOTO 23009
C
C3008   CONTINUE
23008   CONTINUE
C
C       TIME = 0
        TIME = 0
C
```

```
C3009  CONTINUE
23009  CONTINUE
C
C3006  IF(.NOT.(TIME .EQ. 2)) GOTO 23005
23006  IF(.NOT.(TIME .EQ. 2)) GOTO 23005
C
C3007  CONTINUE
23007  CONTINUE
C
C      T = T0 + (T1 - T0) * (X + (2.0/PI) * SUM)
C =====  MIXED MODE OPERANDS ACCEPTED =====
       CALL MVSUB (T1,T0,MVTMP(1,1))
       CALL MVCRM (2.0,MVTMP(1,2))
       CALL MVDIV (MVTMP(1,2),PI,MVTMP(1,2))
       CALL MVMUL (MVTMP(1,2),SUM,MVTMP(1,2))
       CALL MVADD (X,MVTMP(1,2),MVTMP(1,2))
       CALL MVMUL (MVTMP(1,1),MVTMP(1,2),MVTMP(1,2))
       CALL MVADD (T0,MVTMP(1,2),T)
C
C       WRITE (6,4) FEET
        WRITE (6,4) FEET
C
C       CALL MVPUT (T)
        CALL MVPUT (T)
4        FORMAT (1X, 15HTHE ANSWER FOR ,I2, 9H FEET IS )
C
C       WRITE (6,5)
        WRITE (6,5)
5        FORMAT (25H The number of steps is: )
C
C       CALL MVPUT (COUNT)
        CALL MVPUT (COUNT)
C
C3003  CONTINUE
23003  CONTINUE
C
C3004  CONTINUE
23004  CONTINUE
C
C       CALL CTP(0)
        CALL CTP(0)
C
C       STOP
        STOP
        END
```

Appendix G

User Guide to Structured Variable Precision Interval FORTRAN

1. General Description of the Package Design Methodology

The variable precision interval arithmetic package contains three distinct levels of subroutines. The first level is called by the user program. This first level of routines is distinguished by the prefix MV. The first level addition routine, for example, is called MVADD. This level is responsible for controlling the precision of the computation. Its duties include:

1) making adjustments to the precision of the arguments so that when they are passed to the next level they are of the same precision.

2) determining and passing the precision in which the operation is to be performed to the underlying routines. This is done via an external variable in unlabelled common.

3) the passing of the argument values to the next level in a form acceptable to the underlying routines (the format of the multiple precision value is somewhat different at this level than below).

4) examining the target of the computation and assigning the value of the result to the target with appropriate adjustments to the precision of the result.

The second level, with name prefix MX, is responsible for the interval arithmetic aspects of the computation. Its duties include:

1) case analysis (when necessary) to determine which endpoints of the arguments are to be used in the

computation.

2) the passing of the appropriate endpoints to the next level with a proper indication of the rounding strategy to be employed (care is taken that no overwriting of values occurs so that computations like A = A + A are acceptable).

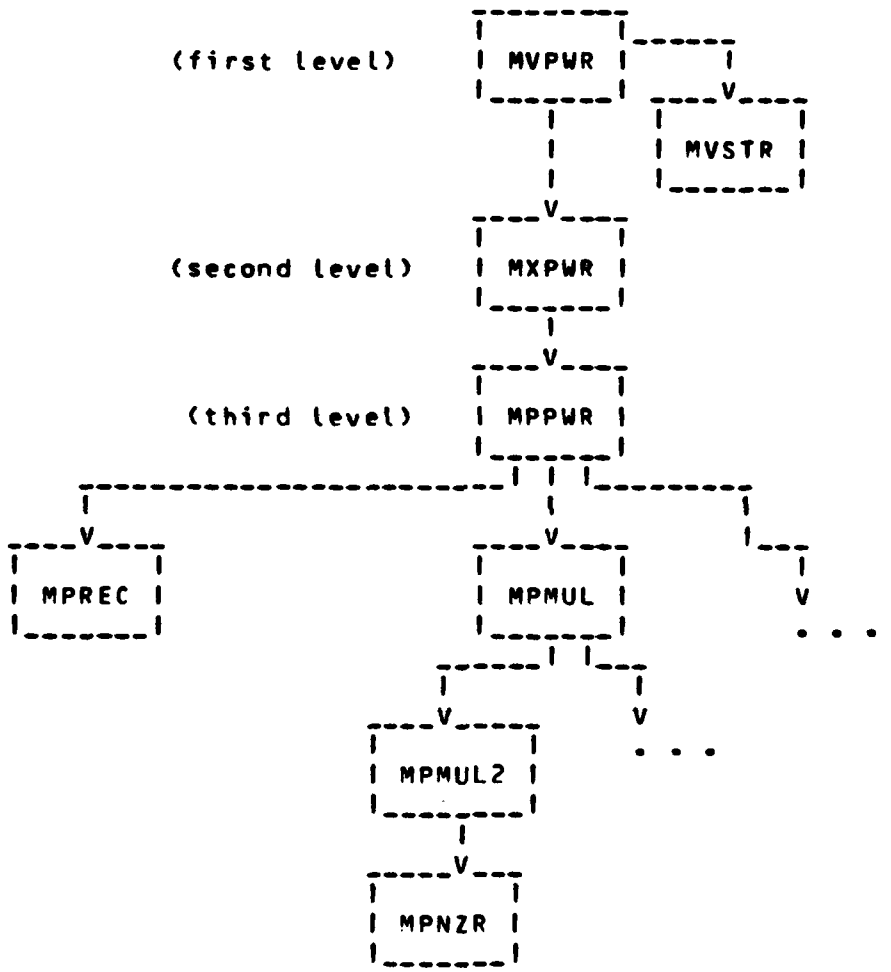


FIGURE 39

Tree Figure of the multilevel interpretive structure of the power function

The third level, with name prefix MP, is responsible for the actual performance of the computation employing the rounding strategy received as a parameter from the second level. For example, the subroutine invocation history of X = Y**N is graphically represented is Figure 39.

Within the package there are two slightly different multiple precision interval "word" formats used. The third level operates on one endpoint of the second level interval. The format used at the third level is the same as either the left or right portions of the second level interval format.

The first level interval format, Figure 40, is used only at the first level. The multiple precision word is represented as an integer array with the first element carrying three pieces of information. The first digit is used to represent the sign -- +1 signifies positive, -1 negative, 0 zero. The three digits following the sign is the precision of the word. The next digit determines whether or not the word is a temporary variable. 1 is used to indicate a temporary; two is used to indicate a non-temporary. For zero the exponent and digits are undefined. The second element carries the exponent of the multiple precision word. The exponent is always a base 10 integer and signifies (base)**expon. The next N elements contain the N digits of the N digit precision number. The digits can be of any base (with some restriction on size) but the package was implemented with base 10

as a matter of convenience. Using base 100 would result in almost a 50% savings in the amount of space used by each variable precision interval variable, but would require some slight modifications to the input/output routines.

FIGURE 40

Multiple precision interval number format. for first level, sdddt = sign(0,-1 or +1) concatenated with the precision concatenated with a temporary variable indicator (1 for temporary, 0 for non-temporary) expon = exponent (to base b) Ti = digit (in base b) s = sign (0, -1 or +1). Note that expon and Ti are undefined if sign = 0.

The second level interval format, Figure 41, is used at the second level. The multiple precision interval word at this level is identical to that of the first level except for the deletion of the precision information and the temporary variable indicator.

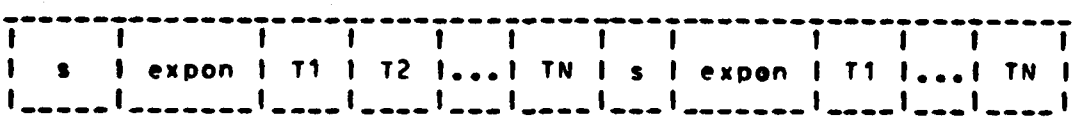


FIGURE 41

Multiple precision interval number format. for second level. s = sign(0,-1 or +1). Note that the exponent and digits are undefined if sign = 0 expon = exponent (to base b) Ti = digit (in base b)

2. Description of the MV Routines

The MV routines can be divided into two classes -- one argument and two argument routines. Within each class all MV routines are virtually identical. The partitioning of the two argument routines is given in Figure 42.

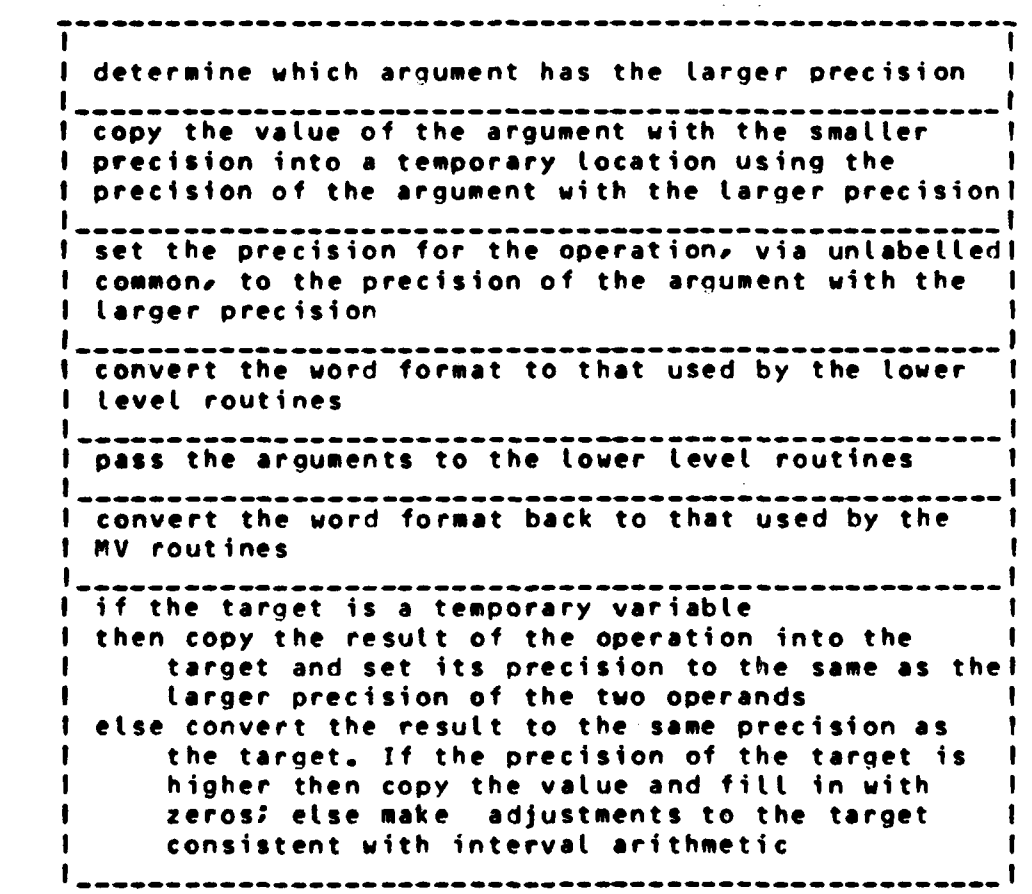


FIGURE 42

Partioning of a two argument MV routine

The one argument routines are essentially the same; however,

there is clearly no need for the precision manipulations necessary for the two argument routines. The MV routines available are described in Appendix E.

3. Preparation of the User Program

3.1 RATFOR and AUGMENT Essentials

The translation sequence from structured variable precision interval FORTRAN to standard FORTRAN is done in two passes. On the first pass the program is converted from RATFOR format to standard FORTRAN format. It is necessary that the program be written in RATFOR for the conversion. A summary of RATFOR is given in Appendix A with sample programs in Appendix F.

On the second pass an AUGMENT description deck, described in Appendix B, is attached to the user program and processed through the AUGMENT precompiler. AUGMENT converts the variable precision interval variables and the operations upon them to standard FORTRAN. This also is discussed in Appendix B.

3.2 Requirements for the Use of Variable Precision Interval Variables

Several requirements must be met by the user for the use of variable precision interval variables.

1) variable precision interval variables be declared VARINTERVAL.

2) the string "%C*BEGIN" must be placed in column 1 after the declarative statement but before the first executable statement.

3) there must be FORTRAN DATA statements setting the value of the first element of each variable precision interval variable to the precision that it is to contain.

4) the program must be written in RATFOR which is described in Appendix A.

Suppose the user were to have three variable precision interval variables X, Y, and Z with precisions 10, 20 and 30 respectively. The program in Figure 43 would be a valid example of a program containing VARINTERVAL variables. Figure 44 through Figure 46 are examples of invalid VARINTERVAL programs.

```
      INTEGER KT
      DATA X, Y, Z /10,20,30/
      VARINTERVAL X, Y, Z
%C*BEGIN
# VALID RATFOR COMMENT
      CALL MVGET (X)
      READ (5,1) KT
1     FORMAT (I10)
      IF (X >= 10.7)
        [Y = X / 18.3
         Z = KT+Y ]
      ELSE
        [Z = X * 18.3 ]
      CALL MVPUT (Z)
      STOP
      END
```

FIGURE 43

Example of a valid VARINTERVAL program

```
      VARINTERVAL X, Y
      VARINTERVAL Z
      INTEGER KT
      DATA X(1),Y(1),
                Z(1) /10,20,30/
# VALID RATFOR COMMENT
      CALL MVGET (X)
      READ (5,1) KT
1     FORMAT (I10)
      IF (X >= 10.7)
        [Y = X / 18.3
         Z = KT+Y ]
      ELSE
        [Z = X * 18.3 ]
      CALL MVPUT (Z)
      STOP
      END
```

FIGURE 44

Example of a invalid VARINTERVAL program
(missing %C*BEGIN)

```
      INTEGER KT
      VARINTERVAL X,Y,Z
%C*BEGIN
# VALID RATFOR COMMENT
      CALL MVGET (X)
      READ (5,1) KT
1     FORMAT (I10)
      IF (X >= 10.7)
        [Y = X / 18.3
         Z = KT+Y ]
      ELSE
        [Z = X * 18.3 ]
      CALL MVPUT (Z)
      STOP
      END
```

FIGURE 45

Example of a invalid VARINTERVAL program
(no DATA statement setting precision)

```
      INTEGER KOUNT
      DATA X(1),Y(1),Z(1)/10,20,30/
      VARINTERVAL X,Y,Z
C THIS IS NOT A RATFOR COMMENT
      IF (X.GE.10.7) GO TO 10
      X = X * 18.3
      KOUNT = KOUNT - 1
      GO TO 20
10    X = X / 18.3
      KOUNT = KOUNT + 1
20    CONTINUE
      STOP
      END
```

FIGURE 46

Example of invalid VARINTERVAL program (not written in RATFOR)

3.3 Inputting and Outputting of Variable Precision Interval Variables

There is very little for the user to be concerned with during the translation of a structured variable precision interval FORTRAN program to standard FORTRAN. RATFOR and AUGMENT take care of almost all of the details. The major exception is that i/o statements are ignored during the translation process. It is left up to the user to make the necessary modifications to the program before translation so that i/o is done properly.

For the input of variable precision interval values the subroutine MVGET is supplied. Its usage is

CALL MVGET (variable precision interval variable)

which will read a variable precision interval value from the FORTRAN logical unit specified by LUN. Each endpoint must be on a separate line with the left endpoint appearing first. Generalized floating point format is accepted by the routine. The input is checked to insure that it forms a proper interval. A warning is output if the number of digits in the input is greater than the precision of the target. Truncation of the input to the precision of the target occurs in this case.

For the outputting of variable precision interval values the subroutine MVPUT is supplied. Its usage is

CALL MVPUT (variable precision interval variable)

which will output the specified value to the FORTRAN logical unit specified by LUN. Output is one value to a line.

3.4 Constraints on Variable Precision Interval Usage

There are, naturally, constraints. These are:

1) The precision of a variable precision interval variable must be less than 1000 but more than 2.

2) The maximum exponent range is 2000.

3) Unlabelled common must not be used by the user or it will disturb the lower levels of the support structure.

4) Avoid using any names beginning with MV, MX or MP.

Additional information concerning these constraints may be found in Appendix C and Appendix E.

3.5 Using the Varint Command on Multics

The command for the invocation of the virtual compiler for the translation of structured variable precision FORTRAN is "varint" or "vt".

The syntax is:

```
vt path -maxp N {-control_args}
```

The arguments are as follows:

path
: is the pathname of a FORTRAN source segment; the fortran suffix need not be given.

-maxp N
: is a mandatory control argument which indicates to the virtual compiler that the maximum precision that will be encountered in the program will be <= N, 2<N<1000.

Control arguments:

-no_compile, -nc
: does not produce an object code segment from the FORTRAN output.

-no_translated_source, -nt
: checks the VARINTERVAL program for correctness but does not produce the FORTRAN output.

-augment_list, als
: produce an AUGMENT listing of the FORTRAN output.

Any valid FORTRAN compiler option.

3.6 Practical Comments

During the course of testing and development there were encountered several instances of the use of VARINTERVAL variables in which the way in which the algorithm was configured seriously affected the results that were produced. This is not unusual in itself; such effects can be noted in all computer implementations of algorithms. There were, however, several effects peculiar to VARINTERVAL usage. The user must have a firm understanding of these effects produced by the configuration of his algorithm. With such an understanding the user can ensure that the algorithm is implemented in an efficient manner and produces the proper results. Without such an understanding he may produce a highly inefficient FORTRAN program and unknowingly encounter deleterious side effects. The effects associated with VARINTERVAL usage fall into three categories:

1) effects associated with the use of standard FORTRAN variables and constants

2) effects associated with attempts at precision control

3) effects associated with the repetitious evaluation of invariant expressions

The following discussion describes in detail what problems are encountered with these effects and how the user may avoid them.

The usage of standard FORTRAN variables can create serious

problems for the unwary. These problems can be summarized as:

1) induction of false significance

2) loss of significance

3) repetitious conversions to VARINTERVAL

False significance can be induced into the results by the mixing of standard FORTRAN variables with VARINTERVAL variables. The problem occurs when a VARINTERVAL value of low significance is assigned to a standard FORTRAN variable and then reassigned to a VARINTERVAL variable. When a VARINTERVAL value is assigned to a standard FORTRAN variable the midpoint of the VARINTERVAL value is used. Standard FORTRAN values are taken as exact results when they are assigned to VARINTERVAL variables. Figure 47 clearly illustrates the problem that can arise.

```
REAL Y
VARINTERVAL X, Z

# value input is [-150, +150]
CALL MVGET (X)

# value assigned to Y is 0.0
Y = X

# Z is given the value [0,0]
Z = Y
```

FIGURE 47

Example of induction of false significance

A loss of significance may also appear when standard FORTRAN and VARINTERVAL variables are mixed. This loss of significance takes

place in the same manner as the induction of false significance. The problem crops up when a VARINTERVAL value of high significance is assigned to a standard FORTRAN variable which has, at most, 8 to 20 decimal digits of precision. Figure 48 demonstrates the problem that might occur.

```
REAL Y
VARINTERVAL X, Z

# value input is [1/3,1/3] to 50 decimal
        # digits of precision
CALL MVGET (X)

# value assigned is .333333333
Y = X

# value assigned is [.33333333,.333333333]
Z = Y
```

FIGURE 48

Example of loss of significance

Loss of significance may also be induced through the use of constants. This may occur in two ways. The first is through the use of inappropriately insignificant constants. For example, the use of the constant pi to 18 digits of accuracy in a section of code which is to be performed with 100 digits of precision is a waste of resources. The second manner in which loss of significance may occur is through the conversions which naturally occur with real constants. Suppose the program contains the statement

X = .3456

where X is VARINTERVAL with 30 digits of precision. The value of

X after the assignment is not .3456 but

0.345600003004074096679687500000e+000

The reason for the inexact conversion is that .3456 is not exactly representable in binary, to which it is converted at compile time. This inexact binary value is converted to VARINTERVAL, resulting in an overall inexact conversion. The user can avoid the problem by avoiding real constants. If the program contained, instead, the two statements

```
X = 3456
X = X/10000
```

the conversion will be exact.

The problems associated with the use of constants can be avoided with the use of care. The user must be especially alert when converting an existent program to VARINTERVAL. It is quite easy to overlook constants that may cause problems. The consequences are grave; these validity of the program's results may be seriously affected. Furthermore, these problems are not the type which are detected by the VARINTERVAL data type. The results produced by the program can appear entirely satisfactory even when seriously in error. The third problem that the user might encounter when mixing VARINTERVAL and standard FORTRAN variables is one of efficiency. In some situations this mixing will have no effect on the significance of the result, for example, the use of a DO index in the body of a loop. There may, however, be some objectionable expense incurred through multiple conversions of the same value to the same data type. The program segment in

Figure 49 exhibits this behavior.

```
INTEGER I
VARINTERVAL X, Y, Z

# This loop performs three conversions of I
#      to VARINTERVAL
 DO I = 1,  100
   [ X = I
     Y = I
     X = I ]
```

FIGURE 49

Example of repetitious conversions

The user must take care when attempting to exert control over the precision of the operations. Consider, for example, the statement

T = X * Y * Z

where each variable is VARINTERVAL with 50 digits of precision. The user may decide that the precision necessary for the derivation of an acceptable result is 20 digits of precision. There are two considerations the user must make when lowering the precision of a section of code. The first of these is that the precision is propagated by intermediate results. Thus, if T, X and Z have a precision of 20 while Y has a precision of 50, the entire calculation will be performed at precision 50. The user must be careful that such unwanted precision propagation does not occur. The other consideration pertains to carelessness. The user must guard against the lowering of a variables precision when that variables is used elsewhere in the program, thus unintentionally affecting the precision of other results.

Another aspect of precision control that is easily overlooked is the adjustment of loop termination values. Suppose a program contains the loop shown in Figure 50, where all variables are VARINTERVAL with a precision of 500 decimal digits. In a normal FORTRAN program ERROR would typically have a value of no less than 1.0e-15. This is entirely inadequate for the current situation; the result would still be significant to only 15 or 20 decimal digits. A value of 1.0e-400 would be a much more appropriate value.

```
REPEAT
  [ COUNT = COUNT + 1
    XSUM = EXP(COUNT*Y)
    SUM = SUM + XSUM ]
UNTIL (XSUM < ERROR)
```

FIGURE 50

Example where inadequate loop termination value may exist

The third category of effects are those associated with the repetitious evaluation of invariant expressions. When building expressions the user must remember that any operation involving VARINTERVAL operands is non-trivial, especially if performed with high precision. Thus, the compensation for the effort of removing only one or two repetitious operations from an expression is usually justifiable. Figure 51 and Figure 52 give two examples of configurations which contain such invariant expressions and a reconfiguration which is more efficient.

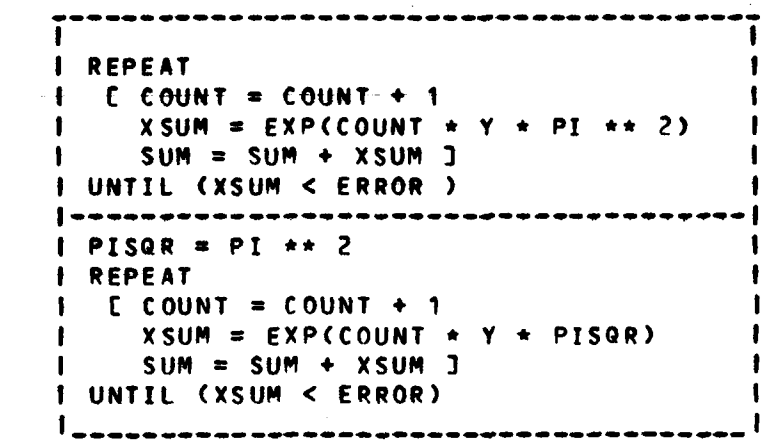


FIGURE 51

Example of invariant expression problem and solution

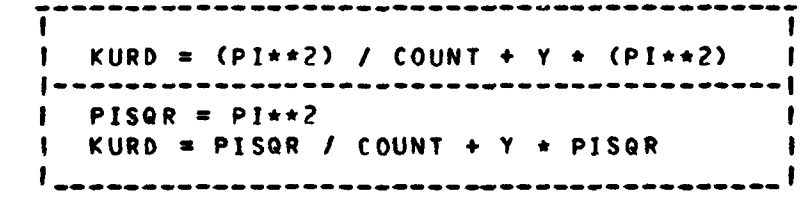


FIGURE 52

Example of invariant expression problem and solution

Appendix H

Calculated value of the constant e to 1000 digits

```
2.71828182845904523536028747135266249775724709369995
957496696762772407663035354759457138217852516642 74
274663919320030599218174135966290435729003342952 60
595630738132328627943490763233829880753195251019 01
157383418793070215408914993488416750924476146066 80
822648001684774118537423454424371075390777449920 69
551702761838606261331384583000752044933826560297 60
673711320070932870912744374704723069697720931014 16
928368190255151086574637721112523897844250569536 96
770785449969967946864454905987931636889230098793 12
773617821542499922957635148220826989519366803318 25
288693984964651058209392398294887933203625094431 17
301238197068416140397019837679320683282376464804 29
531180232878250981945581530175671736133206981125 09
961818815930416903515988885193458072738667385894 22
879228499892086805825749279610484198444363463244 96
848756023362482704197862320900216099023530436994 18
491463140934317381436405462531520961836908887070 16
768396424378140592714563549061303107208510383750 51
011574770417189861068739696552126715468895703503 5(4)
```

Appendix I

User Source Versions of the e Computation Algorithm

Version 1 -- Single Precision Real

```
# THIS IS A PROGRAM TO COMPUTE e
      REAL SUM, NFACT, STEP, ERROR

# INITIALIZATION
      SUM = 0.0
      NFACT = 1.0
      STEP = 1.0
      ERROR = 1.0E-8
      I = 0
# INITIALIZE THE CPU AND PAGING COUNTERS
      CALL CTP (1)

# LOOP THROUGH UNTIL STEP IS LESS THAN OR EQUAL TO ERROR
      REPEAT
        [ SUM = SUM + STEP
          I = I + 1
          NFACT = NFACT * FLOAT(I)
          STEP = 1/NFACT ]
      UNTIL (STEP <= ERROR)

# DISPLAY THE RESULTS
      WRITE(6,2) SUM
2     FORMAT (1X,"e IS EQUAL TO ",F10.8)
      WRITE(6,1) I
1     FORMAT (1h+,T7,"COMPUTED IN ",I3," STEPS")

# PRESENT THE CPU TIME AND PAGING FOR THIS PROCEDURE
      CALL CTP(0)
      STOP
      END
```

Version 2 -- Double Precision Real

```
# THIS IS A PROGRAM TO COMPUTE e
      DOUBLE PRECISION SUM, NFACT, STEP, ERROR

# INITIALIZATION
      SUM = 0.0D0
      NFACT = 1.0D0
      STEP = 1.0D0
      ERROR = 1.0D-18
      I = 0
# INITIALIZE THE CPU AND PAGING COUNTERS
      CALL CTP (1)

# LOOP THROUGH UNTIL STEP IS LESS THAN OR EQUAL TO ERROR
      REPEAT
        [ SUM = SUM + STEP
          I = I + 1
          NFACT = NFACT * DBLE(FLOAT(I))
          STEP = 1/NFACT ]
      UNTIL (STEP <= ERROR)

# DISPLAY THE RESULTS
      WRITE(6,2) SUM
2     FORMAT (1X,"e IS EQUAL TO ",F20.18)
      WRITE(6,1) I
1     FORMAT (1h+,T7,"COMPUTED IN ",I3," STEPS")

# PRESENT THE CPU TIME AND PAGING FOR THIS PROCEDURE
      CALL CTP(0)
      STOP
      END
```

Version 3 -- Single Precision Interval

```
# THIS IS A PROGRAM TO COMPUTE e
      INTERVAL SUM, NFACT, STEP, ERROR

# INITIALIZATION
      SUM = 0.0
      NFACT = 1.0
      STEP = 1.0
      ERROR = 1.0E-8
      I = 0
# INITIALIZE THE CPU AND PAGING COUNTERS
      CALL CTP (1)

# LOOP THROUGH UNTIL STEP IS LESS THAN OR EQUAL TO ERROR
      REPEAT
        [ SUM = SUM + STEP
          I = I + 1
          NFACT = NFACT * FLOAT(I)
          STEP = 1/NFACT ]
      UNTIL (STEP <= ERROR)

# DISPLAY THE RESULTS
      WRITE(6,2) SUM
2     FORMAT (1X,"e IS EQUAL TO ","[",F10.8,",",F10.8,"]")
      WRITE(6,1) I
1     FORMAT (1h+,T7,"COMPUTED IN ",I3," STEPS")

# PRESENT THE CPU TIME AND PAGING FOR THIS PROCEDURE
      CALL CTP(0)
      STOP
      END
```

Version 4 -- 56 Decimal Digit Interval

```
# THIS IS A PROGRAM TO COMPUTE e
      INTERVAL SUM, NFACT, STEP, ERROR

# INITIALIZATION
      SUM = 0
      NFACT = 1
      STEP = 1
      ERROR = 1
      DO I = 1, 10 # SET ERROR = 1.0e-50
       [ ERROR = ERROR/100000
       ]
      I = 0
# INITIALIZE THE CPU AND PAGING COUNTERS
      CALL CTP (1)

# LOOP THROUGH UNTIL STEP IS LESS THAN OR EQUAL TO ERROR
      REPEAT
        [ SUM = SUM + STEP
          I = I + 1
          NFACT = NFACT * I
          STEP = 1/NFACT ]
      UNTIL (STEP <= ERROR)

# DISPLAY THE RESULTS
      WRITE(6,2)
2     FORMAT (1X,"e IS EQUAL TO")
      CALL INTPRV (" ",1,3,128,SUM)
      WRITE(6,1) I
1     FORMAT (1h+,T7,"COMPUTED IN ",I3," STEPS")

# PRESENT THE CPU TIME AND PAGING FOR THIS PROCEDURE
      CALL CTP(0)
      STOP
      END
```

Version 5 -- Variable Precision Interval

```
# THIS IS A PROGRAM TO COMPUTE e
      DATA SUM(1), NFACT(1), STEP(1), ERROR(1) /4*56/
      VARINTERVAL SUM, NFACT, STEP, ERROR

# INITIALIZATION
      SUM = 0
      NFACT = 1
      STEP = 1
      ERROR = 1
      DO I = 1, 10 # SET ERROR = 1.0e-50
       [ ERROR = ERROR/100000
       ]
      I = 0
# INITIALIZE THE CPU AND PAGING COUNTERS
      CALL CTP (1)

# LOOP THROUGH UNTIL STEP IS LESS THAN OR EQUAL TO ERROR
      REPEAT
        [ SUM = SUM + STEP
          I = I + 1
          NFACT = NFACT * I
          STEP = 1/NFACT ]
      UNTIL (STEP <= ERROR)

# DISPLAY THE RESULTS
      WRITE(6,2)
2     FORMAT (1X,"e IS EQUAL TO")
      CALL MVPUT (SUM)
      WRITE(6,1) I
1     FORMAT (1h+,T7,"COMPUTED IN ",I3," STEPS")

# PRESENT THE CPU TIME AND PAGING FOR THIS PROCEDURE
      CALL CTP(0)
      STOP
      END
```

Appendix J

User Source Versions of the Heat Computation Algorithm

Version 1 -- Single Precision Real

```
  # THIS IS A PROGRAM WHICH OUTPUTS THE TEMPERATURE OF A PIECE OF
  #    PIPE AT ONE FOOT INTERVALS. AMBIENT TEMPERATURE IS 70
  #    DEGREES; HEAT SOURCE TEMPERATURE IS 500 DEGREES. INPUT IS
  #    THE AMOUNT OF TIME AFTER WHICH END OF THE PIPE HAS BEEN
  #    PLACED AGAINST THE HEAT SOURCE

  REAL PI, K, RO, C, TO, T1, TOUT,
                          THETA, T, X, SUM, XSUM, ERROR

  INTEGER FEET, COUNT, TIME, L

#
#
# BEGIN PROGRAM

  # INITIALIZATIONS
  CALL CTP(1)       #  INITIALIZE THE CPU AND PAGING COUNTERS
  PI = 3.1415927
  L = 10
  K = 30.0
  RO = 7.1 * 62.3
  C = .12
  TO = 70.00
  T1 = 500.0
  ERROR = 1.0E-8


 # INPUT THE AMOUNT OF TIME
  WRITE (6,1)
1 FORMAT (" ENTER THE TIME IN MINUTES")
  READ (5,2) THETA
2 FORMAT (V)

 # LOOP ONCE THROUGH THE COMPUTATION FOR EACH FOOT
  DO FEET = 1, L
    [
     # INITIALIZATIONS FOR INNER LOOP
     X = FLOAT(FEET)/FLOAT(L)   # COMPUTE DISTANCE TO HEAT SOURCE

     # COMPUTE A STARTING VALUE FOR T
     T = (K*THETA)/(FLOAT(L)**2 * RO * C)
```

```
      SUM = 0.0
      XSUM = 0.0
      COUNT = 0
      TIME = 0

      # LOOP TO COMPUTE THE INTERMEDIATE VALUES FOR THE
      #         TEMPERATURE AT THIS DISTANCE
      REPEAT
        [
         COUNT = COUNT + 1
         XSUM =  ((-1)**COUNT/FLOAT(COUNT)) *
                 (EXP( -(FLOAT(COUNT)**2) * (PI**2) * T) *
                 SIN (FLOAT(COUNT)*PI*X))

         # CHECK FOR INTERMEDIATE VALUE LESS THAN ERROR
         IF (ABS(XSUM) < ERROR)
           [
             # SINCE SIN CAN GO CLOSE TO 0, LET ERROR VALUE
             #     BE EXCEDED TWICE
             TIME = TIME + 1
           ]
         ELSE TIME = 0

         SUM = SUM + XSUM

        ]          # END OF LOOP TO COMPUTE INTERMEDIATE VALUES
      UNTIL (TIME == 2)


      # COMPUTE AND OUTPUT THE TEMPERATURE FOR THIS DISTANCE

       TOUT = T0 + (T1 - T0) * (X + (2.0/PI) * SUM)
       IF (FEET == 1)
         [   WRITE (6,4) FEET, TOUT
           4 FORMAT (1X, "THE ANSWER FOR ",I2, " FOOT IS ",F20.6)
         ]
       ELSE
         [   WRITE (6,5) FEET, TOUT
           5 FORMAT (1X, "THE ANSWER FOR ",I2, " FEET IS ",F20.6)
         ]

    ]  # END OF MAIN ITERATION LOOP

  # OUTPUT THE FINAL CPU AND PAGING VALUES
  CALL CTP(0)

  STOP
  END
```

Version 2 -- Double Precision Real

```
  # THIS IS A PROGRAM WHICH OUTPUTS THE TEMPERATURE OF A PIECE OF
  #     PIPE AT ONE FOOT INTERVALS. AMBIENT TEMPERATURE IS 70
  #     DEGREES; HEAT SOURCE TEMPERATURE IS 500 DEGREES. INPUT IS
  #     THE AMOUNT OF TIME AFTER WHICH END OF THE PIPE HAS BEEN
  #     PLACED AGAINST THE HEAT SOURCE

  DOUBLE PRECISION PI, K, RO, C, TO, T1, TOUT,
                   THETA, T, X, SUM, XSUM, ERROR

  INTEGER FEET, COUNT, TIME, L

#
#
# BEGIN PROGRAM


  # INITIALIZATIONS

  CALL CTP(1)       #  INITIALIZE THE CPU AND PAGING COUNTERS

  PI = 3.141592700
  L = 10
  K = 30.000
  RO = 7.100 * 62.300
  C = .1200
  TO = 70.00000
  T1 = 500.000
  ERROR = 1.00-8


 # INPUT THE AMOUNT OF TIME

  WRITE (6,1)
1 FORMAT (" ENTER THE TIME IN MINUTES")
  READ (5,2) THETA
2 FORMAT (V)


 # LOOP ONCE THROUGH THE COMPUTATION FOR EACH FOOT
  DO FEET = 1, L
    [
     # INITIALIZATIONS FOR INNER LOOP
     X = DBLE(FLOAT(FEET))/DBLE(FLOAT(L)) # COMPUTE DISTANCE TO HEAT SOURCE

     # COMPUTE A STARTING VALUE FOR T
     T = (K*THETA)/(DBLE(FLOAT(L))**2 * RO * C)
     SUM = 0.000
     XSUM = 0.000
     COUNT = 0
```

```
      TIME = 0

      # LOOP TO COMPUTE THE INTERMEDIATE VALUES FOR THE
      #         TEMPERATURE AT THIS DISTANCE
      REPEAT
        [
         COUNT = COUNT + 1
         XSUM =  ((-1)**COUNT/DBLE(FLOAT(COUNT))) *
                 (DEXP( -(DBLE(FLOAT(COUNT))**2) * (PI**2) * T) *
                  DSIN (DBLE(FLOAT(COUNT))*PI*X))

         # CHECK FOR INTERMEDIATE VALUE LESS THAN ERROR
         IF (DABS(XSUM) < ERROR)
           [
             # SINCE SIN CAN GO CLOSE TO 0, LET ERROR VALUE
             #     BE EXCEDED TWICE
             TIME = TIME + 1
           ]
         ELSE TIME = 0

         SUM = SUM + XSUM

        ]          # END OF LOOP TO COMPUTE INTERMEDIATE VALUES
      UNTIL (TIME == 2)


      # COMPUTE AND OUTPUT THE TEMPERATURE FOR THIS DISTANCE

       TOUT = T0 + (T1 - T0) * (X + (2.0D0/PI) * SUM)
       IF (FEET == 1)
         [   WRITE (6,4) FEET, TOUT
           4 FORMAT (1X, "THE ANSWER FOR ",I2," FOOT IS ",D16.11)
         ]
       ELSE
         [   WRITE (6,5) FEET, TOUT
           5 FORMAT (1X, "THE ANSWER FOR ",I2," FEET IS ",D16.11)
         ]

     ]  # END OF MAIN ITERATION LOOP

    # OUTPUT THE FINAL CPU AND PAGING VALUES
    CALL CTP(0)

    STOP
    END
```

Version 3 -- Single Precision Interval

```
      # THIS IS A PROGRAM WHICH OUTPUTS THE TEMPERATURE OF A PIECE OF
      #     PIPE AT ONE FOOT INTERVALS. AMBIENT TEMPERATURE IS 70
      #     DEGREES; HEAT SOURCE TEMPERATURE IS 500 DEGREES. INPUT IS
      #     THE AMOUNT OF TIME AFTER WHICH END OF THE PIPE HAS BEEN
      #     PLACED AGAINST THE HEAT SOURCE

      INTERVAL PI, K, RO, C, TO, T1, TOUT, TEMPL, TEMPCT,
                              THETA, T, X, SUM, XSUM, ERROR

      INTEGER FEET, COUNT, TIME, L

#
#
# BEGIN PROGRAM


      # INITIALIZATIONS

      CALL CTP(1)       #  INITIALIZE THE CPU AND PAGING COUNTERS

      PI = 3.1415927
      L = 10
      TEMPL = L
      K = 30.0
      RO = 7.1 * 62.3
      C = .12
      TO = 70.00
      T1 = 500.0
      ERROR = 1.0E-8


     # INPUT THE AMOUNT OF TIME

      WRITE (6,1)
    1 FORMAT (" ENTER THE TIME IN MINUTES")
      READ (5,2) THETA
    2 FORMAT (V)


     # LOOP ONCE THROUGH THE COMPUTATION FOR EACH FOOT
      DO FEET = 1, L
        [
         # INITIALIZATIONS FOR INNER LOOP
         X = FLOAT(FEET)/TEMPL    # COMPUTE DISTANCE TO HEAT SOURCE

         # COMPUTE A STARTING VALUE FOR T
         T = (K*THETA)/(TEMPL**2 * RO * C)
         SUM = 0.0
         XSUM = 0.0
```

```
      COUNT = 0
      TIME = 0

      # LOOP TO COMPUTE THE INTERMEDIATE VALUES FOR THE
      #         TEMPERATURE AT THIS DISTANCE
      REPEAT
        [
         COUNT = COUNT + 1
         TEMPCT = COUNT
         XSUM =  ((-1)**COUNT/TEMPCT) *
                 (EXP( -(TEMPCT**2) * (PI**2) * T) *
                 SIN (TEMPCT * PI * X))

         # CHECK FOR INTERMEDIATE VALUE LESS THAN ERROR
         IF (ABS(XSUM) < ERROR)
           [
             # SINCE SIN CAN GO CLOSE TO 0, LET ERROR VALUE
             #    BE EXCEDED TWICE
             TIME = TIME + 1
           ]
         ELSE TIME = 0

         SUM = SUM + XSUM

        ]          # END OF LOOP TO COMPUTE INTERMEDIATE VALUES
      UNTIL (TIME == 2)


      # COMPUTE AND OUTPUT THE TEMPERATURE FOR THIS DISTANCE

       TOUT = T0 + (T1 - T0) * (X + (2.0/PI) * SUM)
       IF (FEET == 1)
         [   WRITE (6,4) FEET, TOUT
           4 FORMAT (1X, "THE ANSWER FOR ",I2," FOOT IS ",
                     "[",F11.6,",",F11.6,"]")
         ]
       ELSE
         [   WRITE (6,5) FEET, TOUT
           5 FORMAT (1X, "THE ANSWER FOR ",I2," FEET IS ",
                     "[",F11.6,",",F11.6,"]")
         ]

     ]  # END OF MAIN ITERATION LOOP

    # OUTPUT THE FINAL CPU AND PAGING VALUES
    CALL CTP(0)

    STOP
    END
```

Version 4 -- 56 Decimal Digit Interval

```
# THIS IS A PROGRAM WHICH OUTPUTS THE TEMPERATURE OF A PIECE OF
#     PIPE AT ONE FOOT INTERVALS. AMBIENT TEMPERATURE IS 70
#     DEGREES; HEAT SOURCE TEMPERATURE IS 500 DEGREES. INPUT IS
#     THE AMOUNT OF TIME AFTER WHICH END OF THE PIPE HAS BEEN
#     PLACED AGAINST THE HEAT SOURCE

INTERVAL PI, K, RO, C, TO, T1, TOUT, TEMPL, TEMPCT,
                    THETA, T, X, SUM, XSUM, ERROR

INTEGER FEET, COUNT, TIME, L

#
#
# BEGIN PROGRAM


  # INITIALIZATIONS

  CALL CTP(1)       #  INITIALIZE THE CPU AND PAGING COUNTERS

  PI = 31415927             # MAKE CONVERSION EXACT
  PI = PI/10000000
  L = 10
  TEMPL = L
  K = 30
  RO = 71 * 623             #  RO = 7.1 * 62.3
  RO = RO/100
  C = 12                    #  C = .12
  C = C/100
  TO = 70
  T1 = 500
  ERROR = 1                 # ERROR = 1.0E-8
  ERROR = ERROR/10000
  ERROR = ERROR/10000


 # INPUT THE AMOUNT OF TIME

  WRITE (6,1)
1 FORMAT (" ENTER THE TIME IN MINUTES")
  CALL INTRDV (THETA, EOF)


 # LOOP ONCE THROUGH THE COMPUTATION FOR EACH FOOT
  DO FEET = 1, L
    {
     # INITIALIZATIONS FOR INNER LOOP
     X = FLOAT(FEET)/TEMPL   # COMPUTE DISTANCE TO HEAT SOURCE

     # COMPUTE A STARTING VALUE FOR T
```

```
      T = (K*THETA)/(TEMPL**2 * RO * C)
      SUM = 0.0
      XSUM = 0.0
      COUNT = 0
      TIME = 0

      # LOOP TO COMPUTE THE INTERMEDIATE VALUES FOR THE
      #        TEMPERATURE AT THIS DISTANCE
      REPEAT
        [
         COUNT = COUNT + 1
         TEMPCT = COUNT
         XSUM =  ((-1)**COUNT/TEMPCT) *
                 (EXP( -(TEMPCT**2) * (PI**2) * T) *
                 SIN (TEMPCT * PI * X))

         # CHECK FOR INTERMEDIATE VALUE LESS THAN ERROR
         IF (ABS(XSUM) < ERROR)
           [
             # SINCE SIN CAN GO CLOSE TO 0, LET ERROR VALUE
             #     BE EXCEDED TWICE
             TIME = TIME + 1
           ]
         ELSE TIME = 0

         SUM = SUM + XSUM

        ]          # END OF LOOP TO COMPUTE INTERMEDIATE VALUES
      UNTIL (TIME == 2)


      # COMPUTE AND OUTPUT THE TEMPERATURE FOR THIS DISTANCE

       TOUT = T0 + (T1 - T0) * (X + (2.0/PI) * SUM)
       IF (FEET == 1)
         [    WRITE (6,4) FEET
           4  FORMAT (1X, "THE ANSWER FOR ",I2, " FOOT IS ")
              CALL INTPRV (" ",1,3,128,TOUT)
         ]
       ELSE
         [    WRITE (6,5) FEET
           5  FORMAT (1X, "THE ANSWER FOR ",I2, " FEET IS ")
              CALL INTPRV (" ",1,3,128,TOUT)
         ]

      ]  # END OF MAIN ITERATION LOOP

    # OUTPUT THE FINAL CPU AND PAGING VALUES
    CALL CTP(0)
    STOP
    END
```

Version 5 -- Variable Precision Interval

```
  # THIS IS A PROGRAM WHICH OUTPUTS THE TEMPERATURE OF A PIECE OF
  #    PIPE AT ONE FOOT INTERVALS. AMBIENT TEMPERATURE IS 70
  #    DEGREES; HEAT SOURCE TEMPERATURE IS 500 DEGREES. INPUT IS
  #    THE AMOUNT OF TIME AFTER WHICH END OF THE PIPE HAS BEEN
  #    PLACED AGAINST THE HEAT SOURCE

  # INTEGER VARIABLES

  INTEGER FEET, COUNT, TIME, L


  # MULTIPLE PRECISION INTERVAL VARIABLES

  DATA PI(1), K(1), RO(1), C(1), TO(1), T1(1), TOUT(1),
       TEMPL(1), TEMPCT(1), THETA(1), T(1), X(1), SUM(1),
       XSUM(1), ERROR(1) /15*56/
  VARINTERVAL PI, K, RO, C, TO, T1, TOUT, TEMPL, TEMPCT,
                          THETA, T, X, SUM, XSUM, ERROR


%C*BEGIN
#
#
# BEGIN PROGRAM


  # INITIALIZATIONS

  CALL CTP(1)       #  INITIALIZE THE CPU AND PAGING COUNTERS

  PI = 31415927      # MAKE CONVERSION EXACT
  PI = PI/10000000
  L = 10
  TEMPL = L
  K = 30
  RO = 71 * 623      # RO = 7.1 * 62.3
  RO = RO/100
  C = 12
  C = C/100
  TO = 70
  T1 = 500
  ERROR = 1
  ERROR = ERROR/10000
  ERROR = ERROR/10000


 # INPUT THE AMOUNT OF TIME

  WRITE (6,1)
```

```
1 FORMAT (" ENTER THE TIME IN MINUTES")
  CALL MVGET (THETA)
2 FORMAT (V)


 # LOOP ONCE THROUGH THE COMPUTATION FOR EACH FOOT
  DO FEET = 1, L
    [
     # INITIALIZATIONS FOR INNER LOOP
     X = FLOAT(FEET)/TEMPL   # COMPUTE DISTANCE TO HEAT SOURCE

     # COMPUTE A STARTING VALUE FOR T
     T = (K*THETA)/(TEMPL**2 * RO * C)
     SUM = 0.0
     XSUM = 0.0
     COUNT = 0
     TIME = 0

     # LOOP TO COMPUTE THE INTERMEDIATE VALUES FOR THE
     #         TEMPERATURE AT THIS DISTANCE
     REPEAT
       [
        COUNT = COUNT + 1
        TEMPCT = COUNT
        XSUM =  ((-1)**COUNT/TEMPCT) *
                (EXP( -(TEMPCT**2) * (PI**2) * T) *
                SIN (TEMPCT * PI * X))

        # CHECK FOR INTERMEDIATE VALUE LESS THAN ERROR
        IF (ABS(XSUM) < ERROR)
          [
            # SINCE SIN CAN GO CLOSE TO 0, LET ERROR VALUE
            #    BE EXCEDED TWICE
            TIME = TIME + 1
          ]
        ELSE TIME = 0

        SUM = SUM + XSUM

       ]          # END OF LOOP TO COMPUTE INTERMEDIATE VALUES
     UNTIL (TIME == 2)


     # COMPUTE AND OUTPUT THE TEMPERATURE FOR THIS DISTANCE

      TOUT = T0 + (T1 - T0) * (X + (2.0/PI) * SUM)
      IF (FEET == 1)
        [   WRITE (6,4) FEET
          4 FORMAT (1X, "THE ANSWER FOR ",I2, " FOOT IS ")
            CALL MVPUT (TOUT)
        ]
      ELSE
        [   WRITE (6,5) FEET
```

```
            5 FORMAT (1X, "THE ANSWER FOR ",I2," FEET IS ")
              CALL MVPUT (TOUT)
          ]

        ]  # END OF MAIN ITERATION LOOP

      # OUTPUT THE FINAL CPU AND PAGING VALUES
      CALL CTP(0)

      STOP
      END
```

Version 6 -- Variable Precision Interval with Partial Optimizations

```
  # THIS IS A PROGRAM WHICH OUTPUTS THE TEMPERATURE OF A PIECE OF
  #     PIPE AT ONE FOOT INTERVALS. AMBIENT TEMPERATURE IS 70
  #     DEGREES; HEAT SOURCE TEMPERATURE IS 500 DEGREES. INPUT IS
  #     THE AMOUNT OF TIME AFTER WHICH END OF THE PIPE HAS BEEN
  #     PLACED AGAINST THE HEAT SOURCE

  # INTEGER VARIABLES

  INTEGER FEET, COUNT, TIME, L


  # MULTIPLE PRECISION INTERVAL VARIABLES

  DATA PI(1), K(1), RO(1), C(1),
       TEMPL(1), TEMPCT(1), THETA(1), T(1), X(1), SUM(1),
       XSUM(1), ERROR(1) /12*200/
  DATA TOUT(1), T0(1), T1(1), X20(1), PI20(1), SUM20(1) /6*20/
  VARINTERVAL PI, K, RO, C, T0, T1, TOUT, TEMPL, TEMPCT,
                          THETA, T, X, SUM, XSUM, ERROR,
                          PI20, X20, SUM20


%C*BEGIN
#
#
# BEGIN PROGRAM


  # INITIALIZATIONS

  CALL CTP(1)       #  INITIALIZE THE CPU AND PAGING COUNTERS

  PI = 31415927      # MAKE CONVERSION EXACT
  PI = PI/10000000
  PI20 = PI
  L = 10
  TEMPL = L
  K = 30
  RO = 71 * 623      # RO = 7.1 * 62.3
  RO = RO/100
  C = 12
  C = C/100
  T0 = 70
  T1 = 500
  ERROR = 1
  ERROR = ERROR/10000
  ERROR = ERROR/10000
```

```
 # INPUT THE AMOUNT OF TIME

  WRITE (6,1)
1 FORMAT (" ENTER THE TIME IN MINUTES")
  CALL MVGET (THETA)
2 FORMAT (V)


 # LOOP ONCE THROUGH THE COMPUTATION FOR EACH FOOT
  DO FEET = 1, L
    [
     # INITIALIZATIONS FOR INNER LOOP
     X = FLOAT(FEET)/TEMPL    # COMPUTE DISTANCE TO HEAT SOURCE

     # COMPUTE A STARTING VALUE FOR T
     T = (K*THETA)/(TEMPL**2 * RO * C)
     SUM = 0.0
     XSUM = 0.0
     COUNT = 0
     TIME = 0

     # LOOP TO COMPUTE THE INTERMEDIATE VALUES FOR THE
     #         TEMPERATURE AT THIS DISTANCE
     REPEAT
       [
        COUNT = COUNT + 1
        TEMPCT = COUNT
        XSUM =  ((-1)**COUNT/TEMPCT) *
                (EXP( -(TEMPCT**2) * (PI**2) * T) *
                SIN (TEMPCT * PI * X))

        # CHECK FOR INTERMEDIATE VALUE LESS THAN ERROR
        IF (ABS(XSUM) < ERROR)
          [
            # SINCE SIN CAN GO CLOSE TO 0, LET ERROR VALUE
            #    BE EXCEDED TWICE
            TIME = TIME + 1
          ]
        ELSE TIME = 0

        SUM = SUM + XSUM

       ]          # END OF LOOP TO COMPUTE INTERMEDIATE VALUES
     UNTIL (TIME == 2)


    # COMPUTE AND OUTPUT THE TEMPERATURE FOR THIS DISTANCE

    X20 = X
    SUM20 = SUM
    TOUT = T0 + (T1 - T0) * (X20 + (2.0/PI20) * SUM20)
    IF (FEET == 1)
      [   WRITE (6,4) FEET
```

```
        4 FORMAT (1X, "THE ANSWER FOR ",I2," FOOT IS ")
          CALL MVPUT (TOUT)
      ]
    ELSE
      [   WRITE (6,5) FEET
        5 FORMAT (1X, "THE ANSWER FOR ",I2," FEET IS ")
          CALL MVPUT (TOUT)
      ]

  ]  # END OF MAIN ITERATION LOOP

  # OUTPUT THE FINAL CPU AND PAGING VALUES
  CALL CTP(0)

  STOP
  END
```

Version_Z_--_Variable_Precision_Interval_with_Full_Optimizations

```
      # PROGRAM TO COMPUTE HEAT TRANSFER IN A TEN FOOT IRON BAR
      #  HEAT SOURCE IS 500 DEGREES, AMBIENT TEMPERATURE IS 70
      #  DEGREES. OUTPUT IS TEMPERATURE OF BAR AT ONE FOOT
      #  INTERVALS. INPUT IS TIME AFTER CONTACT WITH HEAT SOURCE
      #  IN MINUTES.
      #
      # INTEGER VARIABLES
           INTEGER FEET, KLENGTH, ITEMP, TIME
      #
      # VARINTERVAL VARIABLES

           # CONSTANT VALUED VARIABLES
           DATA ERROR(1), PI(1), PISQR(1) /3*200/
           DATA L(1), RO(1), C(1), ROC(1), K(1) /5*6/
           DATA TO(1), T1(1), TDIFF(1), PI20(1), TPI20M(1) /5*20/

           # VARIABLE VALUED VARIABLES
           DATA COUNT(1) /200/
           DATA THETA(1), SUM(1), XSUM(1), X(1), T(1), TEMPL(1),
                PISQRT(1), PIX(1) /8*200/
           DATA TOUT(1), X20(1), SUM20(1)  /3*20/

           # VARINTERVAL DECLARATION
           VARINTERVAL PI, PISQR, L, RO, C, K, TO, T1, COUNT,
                       TOUT, THETA, SUM, XSUM, X, T, ERROR,
                       X20, PI20, SUM20, PISQRT, PIX, TINIT,
                       TPI20M, TDIFF, ROC, TEMPL
      #
      #
      # BEGIN PROGRAM

%C*BEGIN

      # INITIALIZATIONS
           CALL CTP(1)    # INITIALIZE THE CPU AND PAGING COUNTERS
           PI = 31415927  # INITIALIZE PI VALUES
           PI = PI/10000000
           PI20 = PI
           TPI20M = 2/PI20
           PISQR = PI * PI
           L = 10          # LENGTH OF BAR
           TEMPL = L
           K = 30
           RO = 71
           RO = RO * 623
           RO = RO/100
           C = 12
           C = C/100
           ROC = RO * C
```

```
      T0 = 70          # AMBIENT TEMPERATURE
      T1 = 500         # TEMPERATURE OF HEAT SOURCE
      TDIFF = T1 - T0

      ERROR = 1        # ERROR FACTOR FOR LOOP TERMINATION
      ERROR = ERROR/10000 # SET ERROR = 1.E-8
      ERROR = ERROR/10000

#  GET THE TIME, IN MINUTES
      WRITE (6,1)
1     FORMAT (" ENTER THE TIME IN MINUTES")
      CALL MVGET (THETA)
#
#  CALCULATE TEMPERATURE FOR EACH FOOT
#
      KLENGTH = L
      # COMPUTE THE INITIAL VALUE FOR T
      T = (K*THETA)/(L**2 * ROC)
      PISQRT = PISQR * T

      DO  FEET = 1, KLENGTH
         [

           # COMPUTE DISTANCE TO HEAT SOURCE
           X = FLOAT(FEET)/TEMPL

           # LOOP FOR THE COMPUTATION OF SUM
           SUM = 0
           XSUM = 0  # INITIAL LOOP VALUES
           COUNT = 0
           TIME = 0
           PIX = PI * X

            REPEAT
              [ COUNT = COUNT + 1
                ITEMP = COUNT
                XSUM =  ((-1)**ITEMP/COUNT)     *
                     EXP( -(COUNT**2) * PISQRT) *
                     SIN (COUNT*PIX)
                IF (ABS(XSUM) < ERROR )
                  TIME = TIME + 1
                ELSE
                  TIME = 0

                SUM = SUM + XSUM
               ]
             UNTIL (TIME == 2)

            # COMPUTE THE TEMPERATURE FOR THIS DISTANCE
             X20 = X
             SUM20 = SUM
             TOUT = T0 + (TDIFF) * (X20 + (TPI20M) * SUM20)
```

```
        # OUTPUT THE FINAL ANSWER FOR THIS FOOT
         IF (FEET == 1)
           [   WRITE (6,4) FEET
             4 FORMAT (1X,"THE ANSWER FOR ",I2," FOOT IS ")
               CALL MVPUT (TOUT)
           ]
         ELSE
           [   WRITE (6,5) FEET
             5 FORMAT (1X,"THE ANSWER FOR ",I2," FEET IS ")
               CALL MVPUT (TOUT)
           ]
      ]
CALL CTP(0)
STOP
END
```

Appendix K

Results from the Heat Computation Algorithm

```
 THE ANSWER FOR  1 FOOT IS              70.000000
 THE ANSWER FOR  2 FEET IS              70.000000
 THE ANSWER FOR  3 FEET IS              69.999996
 THE ANSWER FOR  4 FEET IS              70.000006
 THE ANSWER FOR  5 FEET IS              70.001100
 THE ANSWER FOR  6 FEET IS              70.072399
 THE ANSWER FOR  7 FEET IS              72.054155
 THE ANSWER FOR  8 FEET IS              95.780487
 THE ANSWER FOR  9 FEET IS             219.179291
 THE ANSWER FOR 10 FEET IS             500.000050
 CPU time =   0.191516 seconds; Page faults = 7
```

Single Precision Real

```
 THE ANSWER FOR  1 FOOT IS       69.999999992
 THE ANSWER FOR  2 FEET IS       69.999999985
 THE ANSWER FOR  3 FEET IS       69.999999999
 THE ANSWER FOR  4 FEET IS       70.000007146
 THE ANSWER FOR  5 FEET IS       70.001103333
 THE ANSWER FOR  6 FEET IS       70.072407058
 THE ANSWER FOR  7 FEET IS       72.054161748
 THE ANSWER FOR  8 FEET IS       95.780486256
 THE ANSWER FOR  9 FEET IS       219.17928843
 THE ANSWER FOR 10 FEET IS       500.00004075
 CPU time = 0.203295 seconds; Page faults = 0
```

Double Precision Real

```
THE ANSWER FOR  1 FOOT IS [  69.999985,  70.000012]
THE ANSWER FOR  2 FEET IS [  69.999976,  70.000024]
THE ANSWER FOR  3 FEET IS [  69.999972,  70.000026]
THE ANSWER FOR  4 FEET IS [  69.999972,  70.000046]
THE ANSWER FOR  5 FEET IS [  70.001081,  70.001131]
THE ANSWER FOR  6 FEET IS [  70.072363,  70.072459]
THE ANSWER FOR  7 FEET IS [  72.054106,  72.054229]
THE ANSWER FOR  8 FEET IS [  95.780422,  95.780548]
THE ANSWER FOR  9 FEET IS [ 219.179247, 219.179348]
THE ANSWER FOR 10 FEET IS [ 500.000042, 500.000065]

CPU time =     6.869834 seconds; Page faults =    0
```

Single Precision Interval

```
THE ANSWER FOR  1 FOOT IS
[69.99999999207139976701397547387148136709048054204478 63,
 69.99999999207139976701397547387148136709048054204478 65 ]
THE ANSWER FOR  2 FEET IS
[69.99999998494462176135940688190590521653896688898202 46,
 69.99999998494462176135940688190590521653896688898202 48 ]
THE ANSWER FOR  3 FEET IS
[69.99999999895284346426452482003105629988238753442584 93,
 69.99999999895284346426452482003105629988238753442584 95 ]
THE ANSWER FOR  4 FEET IS
[70.00000714552716282032172155136076630385358086615018 78,
 70.00000714552716282032172155136076630385358086615018 80 ]
THE ANSWER FOR  5 FEET IS
[70.00110333253882011032089064132378393292193344563392 11,
 70.00110333253882011032089064132378393292193344563392 13 ]
THE ANSWER FOR  6 FEET IS
[70.07240705773198160895665277204328357260142962558511 65,
 70.07240705773198160895665277204328357260142962558511 68 ]
THE ANSWER FOR  7 FEET IS
[72.05416174750445534136670770067972755247015433650963 80,
 72.05416174750445534136670770067972755247015433650963 87 ]
THE ANSWER FOR  8 FEET IS
[95.78048625580866151699952314398608502286186084292752 86,
 95.78048625580866151699952314398608502286186084292752 92 ]
THE ANSWER FOR  9 FEET IS
[219.17928842826718404741031708702930876953103669500792 6,
 219.17928842826718404741031708702930876953103669500792 7 ]
THE ANSWER FOR 10 FEET IS
[500.00004075475740367308408291979763018051232721522906 1,
 500.00004075475740367308408291979763018051232721522906 2 ]

CPU time =  1812.559863 seconds; Page faults =       297
```

56 Decimal Digit Interval

```
THE ANSWER FOR  1 FOOT IS
[ 69.999999992071399767013975473871481367090480542044786401 29,
  69.999999992071399767013975473871481367090480542044786406 70 ]
THE ANSWER FOR  2 FEET IS
[ 69.999999984944621761359406881905905216538966888982024683 99,
  69.999999984944621761359406881905905216538966888982024689 57 ]
THE ANSWER FOR  3 FEET IS
[ 69.999999998952843464264524820031056299882387534425849426 94,
  69.999999998952843464264524820031056299882387534425849432 83 ]
THE ANSWER FOR  4 FEET IS
[ 70.000007145527162820321721551360766303853580866150187880 73,
  70.000007145527162820321721551360766303853580866150187886 97 ]
THE ANSWER FOR  5 FEET IS
[ 70.001103332538820110320890641323783932921933445633921168 81,
  70.001103332538820110320890641323783932921933445633921174 13 ]
THE ANSWER FOR  6 FEET IS
[ 70.072407057731981608956652772043283572601429625585116629 12,
  70.072407057731981608956652772043283572601429625585116642 75 ]
THE ANSWER FOR  7 FEET IS
[ 72.054161747504455341366707700679727552470154336509638342 99,
  72.054161747504455341366707700679727552470154336509638397 75 ]
THE ANSWER FOR  8 FEET IS
[ 95.780486255808661516999523143986085022861860842927528834 78,
  95.780486255808661516999523143986085022861860842927528884 07 ]
THE ANSWER FOR  9 FEET IS
[ 219.179288428267184047410317087029308769531036695007926483 0,
  219.179288428267184047410317087029308769531036695007926494 7 ]
THE ANSWER FOR 10 FEET IS
[ 500.000040754757403673084082919797630180512327215229061100 0,
  500.000040754757403673084082919797630180512327215229061112 9 ]

CPU time =    325.1 seconds;        Page faults =           2
```

56 Digit Variable Precision Interval

```
THE ANSWER FOR  1 FOOT IS
 [ 69.999999992071399767013975473871481367090480542044786403
      892314281549541612132931236791395972874965929O          ,
   69.999999992071399767013975473871481367090480542044786403
      892314281549541612132931236791395972874966015S          ]
THE ANSWER FOR  2 FEET IS
 [ 69.999999984944621761359406881905905216538966888982024686
      865340397050893784634016067252099798613815304S          ,
   69.999999984944621761359406881905905216538966888982024686
      865340397050893784634016067252099798613815421O          ]
THE ANSWER FOR  3 FEET IS
 [ 69.999999998952843464264524820031056299882387534425849429
      886375368473570686183198390829090678100689776S          ,
   69.999999998952843464264524820031056299882387534425849429
      886375368473570686183198390829090678100689902S          ]
THE ANSWER FOR  4 FEET IS
 [ 70.000007145527162820321721551360766303853580866150187883
      142443780009269957170606113310251446118882440l          ,
   70.000007145527162820321721551360766303853580866150187883
      142443780009269957170606113310251446118882588S          ]
THE ANSWER FOR  5 FEET IS
 [ 70.001103332538820110320890641323783932921933445633921170
      940078860058706121438739514221480068719114707S          ,
   70.001103332538820110320890641323783932921933445633921170
      940078860058706121438739514221480068719114830S          ]
THE ANSWER FOR  6 FEET IS
 [ 70.072407057731981608956652772043283572601429625585116634
      702414954624060839772920796668729670638435O432          ,
   70.072407057731981608956652772043283572601429625585116634
      702414954624060839772920796668729670638435145l          ]
THE ANSWER FOR  7 FEET IS
 [ 72.054161747504455341366707700679727552470154336509638369
      846803548672708839144511985712650491937570170S          ,
   72.054161747504455341366707700679727552470154336509638369
      846803548672708839144511985712650491937570800l          ]
THE ANSWER FOR  8 FEET IS
 [ 95.780486255808661516999523143986085022861860842927528858
      799004140045476248063721828254308369645216692O          ,
   95.780486255808661516999523143986085022861860842927528858
      799004140045476248063721828254308369645217322S          ]
THE ANSWER FOR  9 FEET IS
 [ 219.17928842826718404741031708702930876953103669500792649
      232834440560582431255871087304806578487153971O2          ,
   219.17928842826718404741031708702930876953103669500792649
      232834440560582431255871087304806578487153987S4          ]
THE ANSWER FOR 10 FEET IS
 [ 500.000040754757403673084082919797630180512327215229061l0
      976508171491436832419731520033194268778207656700          ,
   500.000040754757403673084082919797630180512327215229061l0
      976508171491436832419731520033194268778207686800          ]
+-----------------------------------------------------------------+
| CPU time =    726.0 seconds;       Page faults =          2     |
+-----------------------------------------------------------------+
```

100 Digit Variable Precision Interval

```
THE ANSWER FOR  1 FOOT IS
 [ 69.999999992071399767013975473871481367090480542044786403 8
      923142815495416121329312367913959728749659720107442516 9
      142294766054028358320475357613957007900761716756073014 6
      698898833016881162444168569419711 64                    ,
   69.999999992071399767013975473871481367090480542044786403 8
      923142815495416121329312367913959728749659720107442516 9
      142294766054028358320475357613957007900761716756073014 6
      698898833016881162444168569419719 63                    ]
THE ANSWER FOR  2 FEET IS
 [ 69.999999984944621761359406881905905216538966888982024686 8
      653403970508937846340160672520997986138153608845897308 6
      529544000239873866721671461515839651563047950572716179 9
      266348965738469858485601137055752 58                    ,
   69.999999984944621761359406881905905216538966888982024686 8
      653403970508937846340160672520997986138153608845897308 6
      529544000239873866721671461515839651563047950572716179 9
      266348965738469858485601137055760 67                    ]
THE ANSWER FOR  3 FEET IS
 [ 69.999999998952843464264524820031056299882387534425849429 8
      863753684735706861831983908290906781006898360193712609 0
      637193992308421798027631840797817373653717141229403259 6
      707354943647899028638683954193140 25                    ,
   69.999999998952843464264524820031056299882387534425849429 8
      863753684735706861831983908290906781006898360193712609 0
      637193992308421798027631840797817373653717141229403259 6
      707354943647899028638683954193147 67                    ]
THE ANSWER FOR  4 FEET IS
 [ 70.000007145527162820321721551360766303853580866150187883 1
      424437800092699571706061133102514461188825149412221032 9
      631795278851065232585213838384769763388431655122649323 3
      805976492958677072738405719983225 41                    ,
   70.000007145527162820321721551360766303853580866150187883 1
      424437800092699571706061133102514461188825149412221032 9
      631795278851065232585213838384769763388431655122649323 3
      805976492958677072738405719983233 69                    ]
THE ANSWER FOR  5 FEET IS
 [ 70.001103332538820110320890641323783932921933445633921170 9
      400788600587061214387395142214800687191147811068127256 8
      349127962983243235534901585280756645369056293836064953 5
      081424252005603777513985633661477 51                    ,
   70.001103332538820110320890641323783932921933445633921170 9
      400788600587061214387395142214800687191147811068127256 8
      349127962983243235534901585280756645369056293836064953 5
      081424252005603777513985633661484 63                    ]
THE ANSWER FOR  6 FEET IS
 [ 70.072407057731981608956652772043283572601429625585116634 7
      024149546240608397729207966687296706384350614691090866 0
      241828898461660909664869492168001519449470299999535872 3
      001805929864586944027461190286864 41                    ,
   70.072407057731981608956652772043283572601429625585116634 7
      024149546240608397729207966687296706384350614691090866 0
      241828898461660909664869492168001519449470299999535872 3
      001805929864586944027461190286879 18                    ]
```

```
THE ANSWER FOR  7 FEET IS
 [ 72.0541617475044553413667077006797275524701543365096383698
      4680354867270883914451198571265049193757048066408398931
      2813918105650332766831809737749136287632909657279240293
      0109404471993774648184945358709141 9,
   72.0541617475044553413667077006797275524701543365096383698
      4680354867270883914451198571265049193757048066408398931
      2813918105650332766831809737749136287632909657279240293
      0109404471993774648184945358709716 9 ]
THE ANSWER FOR  8 FEET IS
 [ 95.7804862558086615169995231439860850228618608429275288587
      9900414004547624806372182825430836964521696998986069616
      8684134696796763213490875684952633456912843179639814248
      8547317657299366196953614365599225 4,
   95.7804862558086615169995231439860850228618608429275288587
      9900414004547624806372182825430836964521696998986069616
      8684134696796763213490875684952633456912843179639814248
      8547317657299366196953614365599773 1 ]
THE ANSWER FOR  9 FEET IS
 [ 219.1792884282671840474103170870293087695310366950079264923
      2834440560582431255871087304806578487153974831922045413
      1788708514327831632956024739420827777730560508683350216
      0658369098517285379256817951564757 16,
   219.1792884282671840474103170870293087695310366950079264923
      2834440560582431255871087304806578487153974831922045413
      1788708514327831632956024739420827777730560508683350216
      0658369098517285379256817951564769 44 ]
THE ANSWER FOR 10 FEET IS
 [ 500.0000407547574036730840829197976301805123272152290611093
      7650817149143683241973152003319426877820766831310742892
      1122074122072617284752919175416633910948799809071822219
      3373363869892092825989956813205442 30,
   500.0000407547574036730840829197976301805123272152290611093
      7650817149143683241973152003319426877820766831310742892
      1122074122072617284752919175416633910948799809071822219
      3373363869892092825989956813205455 20 ]
-------------------------------------------------------------------
CPU time =  2406.0 seconds;       Page faults =         148
-------------------------------------------------------------------
```

200 Digit Variable Precision Interval

```
THE ANSWER FOR  1 FOOT IS
 [ 69.99999999207139976701397547387148136709048054204478640389
    23142815495416121329312367913959728749659720107442516914 22
    94766054028358320475357613957007900761716756073014669889 88
    33016881162444168569419715590680228000528673515316094148 47
    95204677217983535724741350039620921581803199805942557137 41
    62156307614429261712449887475596671390330801953616983392 60
    13637332153818330135777877517562413306028047476556971218 65
    84238260038016895069834127768599763303790960968123544457 41
    74148762712681026530065004628559895762                    ,
   69.99999999207139976701397547387148136709048054204478640389
    23142815495416121329312367913959728749659720107442516914 22
    94766054028358320475357613957007900761716756073014669889 88
    33016881162444168569419715590680228000528673515316094148 47
    95204677217983535724741350039620921581803199805942557137 41
    62156307614429261712449887475596671390330801953616983392 60
    13637332153818330135777877517562413306028047476556971218 65
    84238260038016895069834127768599763303790960968123544457 41
    74148762712681026530065004628559897611                   ]
THE ANSWER FOR  2 FEET IS
 [ 69.99999998494462176135940688190590521653896688898202468686
    53403970508937846340160672520997986138153608845897308652 95
    44000239873866721671461515839651563047950572716179926634 89
    65738469858485601137055756743457018976962915189933948812 34
    82220312921275771584040781678764737576979487394479468442 98
    90848077124494830995331552362728108783019316723670733843 53
    99476051981605914435170208685493175830382015700609715071 06
    11380369097029596057533986266672315382906975544806236406 06
    68784038524087638900936311953814883001                   ,
   69.99999998494462176135940688190590521653896688898202468686
    53403970508937846340160672520997986138153608845897308652 95
    44000239873866721671461515839651563047950572716179926634 89
    65738469858485601137055756743457018976962915189933948812 34
    82220312921275771584040781678764737576979487394479468442 98
    90848077124494830995331552362728108783019316723670733843 53
    99476051981605914435170208685493175830382015700609715071 06
    11380369097029596057533986266672315382906975544806236406 06
    68784038524087638900936311953814884876                   ]
THE ANSWER FOR  3 FEET IS
 [ 69.99999999895284346426452482003105629988238753442584942988
    63753684735706861831983908290906781006898360193712609063 71
    93992308421798027631840797817373653717141229403259670735 49
    43647899028638683954193144024896756175438737069789514952 99
    98352151482130891031901017377519105732725093955304441273 08
    29128600763601357107101155013684114720130049839861245236 49
    12900355267828759494257856168220129608423714798847430137 70
    97378518564737774725681456625896755872714409808497263332 70
    65680151214745928722365606668824563647                   ,
   69.99999999895284346426452482003105629988238753442584942988
    63753684735706861831983908290906781006898360193712609063 71
    93992308421798027631840797817373653717141229403259670735 49
    43647899028638683954193144024896756175438737069789514952 99
    98352151482130891031901017377519105732725093955304441273 08
    29128600763601357107101155013684114720130049839861245236 49
```

```
     1290035526782875949425785616822012960842371479884743013770
     9737851856473777472568145662589675587271440980849726333270
     6568015121474592872236560666882456519 6 ]
THE ANSWER FOR  4 FEET IS
 [ 70.000007145527162820321721551360766303853580866150187883 14
     2443780009269957170606113310251446118882514941222103296317
     9527885106523258521383838476976338843165512264932338059764
     9295867707273840571998322818586139973923409728312569031690
     5765676878356993227546079879004017769266313498146051316835
     9370584670045221558101295985552829481952977291226407257944
     0856953566697643354417975989005780232456762378561802572415
     4954678143373287301659653789853603618706863124203648199022
     2831685063769237252876559237265970787 0
    70.000007145527162820321721551360766303853580866150187883 14
     2443780009269957170606113310251446118882514941222103296317
     9527885106523258521383838476976338843165512264932338059764
     9295867707273840571998322818586139973923409728312569031690
     5765676878356993227546079879004017769266313498146051316835
     9370584670045221558101295985552829481952977291226407257944
     0856953566697643354417975989005780232456762378561802572415
     4954678143373287301659653789853603618706863124203648199022
     2831685063769237252876559237265970968 2 ]
THE ANSWER FOR  5 FEET IS
 [ 70.001103332538820110320890641323783932921933445633921170 94
     0078860058706121438739514221480068719114781106812725683491
     2796298324323553490158528075664536905629383606495350814242
     5200560377751398563366147976439882015922490066417413176718
     2835976607590796248706993481523562119018836939556684745502
     2951461439201607390673582555120334436721717046032935629354
     5451825314425808293147035433771259720167534964548561603809
     2766108911704320215947749976462523581633935990468415181675
     6619543024846717380564954752317744870 7
    70.001103332538820110320890641323783932921933445633921170 94
     0078860058706121438739514221480068719114781106812725683491
     2796298324323553490158528075664536905629383606495350814242
     5200560377751398563366147976439882015922490066417413176718
     2835976607590796248706993481523562119018836939556684745502
     2951461439201607390673582555120334436721717046032935629354
     5451825314425808293147035433771259720167534964548561603809
     2766108911704320215947749976462523581633935990468415181675
     6619543024846717380564954752317744972 7 ]
THE ANSWER FOR  6 FEET IS
 [ 70.072407057731981608956652772043283572601429625585116634 70
     2414954624060839772920796668729670638435061469109086602418
     2889846166090966486949216800151944947029999953587230018059
     2986458694402746119028687132905996117304059114293276805638
     5603275566565597052107813506320817346722375004473242782680
     1501090482595123755358109761811408763785455531020629392621
     4547444599433448547079578775135822665174682891854485805817
     9691816053354282233493106040002370206447299678750805172266
     0648357813612016403084565157950838896 5
    70.072407057731981608956652772043283572601429625585116634 70
     2414954624060839772920796668729670638435061469109086602418
     2889846166090966486949216800151944947029999953587230018059
     2986458694402746119028687132905996117304059114293276805638
```

```
     560327556656559705210781350632081734672237500447324278268O
     150109048259512375535810976181140876378545553102062939262 1
     454744459943344854707957877513582266517468289185448580581 7
     969181605335428223349310604000237020644729967875080517226 6
     064835781361201640308456515795083909 91                    ]
THE ANSWER FOR  7 FEET IS
 [ 72.054161747504455341366707700679727552470154336509638369 84
     680354867270883914451198571265049193757048066408398931281 3
     918105650332766831809737749136287632909657279240293010940 4
     471993774648184945358709464120386791441237320732767919234 4
     049969731626675289385795620739815227533715057239689288895 6
     157679836566136272127565044503848137284477094617152696124 3
     826547280158243704865318203395018524511563765620654237895 5
     677975311894536075313034455426087381720337156017649639508 3
     492850866751668419814279671924095009 38                    ,
    72.054161747504455341366707700679727552470154336509638369 84
     680354867270883914451198571265049193757048066408398931281 3
     918105650332766831809737749136287632909657279240293010940 4
     471993774648184945358709464120386791441237320732767919234 4
     049969731626675289385795620739815227533715057239689288895 6
     157679836566136272127565044503848137284477094617152696124 3
     826547280158243704865318203395018524511563765620654237895 5
     677975311894536075313034455426087381720337156017649639508 3
     492850866751668419814279671924095075 09                    ]
THE ANSWER FOR  8 FEET IS
 [ 95.780486255808661516999523143986085022861860842927528858 79
     900414004547624806372182825430836964521696998986069616868 4
     134696796763213490875684952633456912843179639814248854731 7
     657299366196953614365599540778128109708843347393302202226 1
     577604216256057719202828332723426229385515029629062838504 3
     003365476332088895432495945916091906246478864053100103059 6
     942994571109814577773806435057209887752509484415878101649 9
     975964395036226800228719644236910202817807985860309395674 3
     833548740218564342651228266723195984 24                    ,
    95.780486255808661516999523143986085022861860842927528858 79
     900414004547624806372182825430836964521696998986069616868 4
     134696796763213490875684952633456912843179639814248854731 7
     657299366196953614365599540778128109708843347393302202226 1
     577604216256057719202828332723426229385515029629062838504 3
     003365476332088895432495945916091906246478864053100103059 6
     942994571109814577773806435057209887752509484415878101649 9
     975964395036226800228719644236910202817807985860309395674 3
     833548740218564342651228266723196047 22                    ]
THE ANSWER FOR  9 FEET IS
 [ 219.179288428267184047410317087029308769531036695007926492 3
     283444056058243125587108730480657848715397483192204541178 8
     708514327831632956024739420827777730560508683350216065836 9
     098517285379256817951564768494112531847337135614230294483 3
     599566496585408242307513495553588841170935156792462642610 6
     325688617701626288802749340655947721494876042305996088489
     888565389167675725288023161955083262900265331204553163450 8
     332591772750592260280657863432041248565056032694910686915 3
     743118966082359517504852194508794637976                    ,
    219.179288428267184047410317087029308769531036695007926492 3
     283444056058243125587108730480657848715397483192204541178 8
```

```
|     708514327831632956024739420827777730560508683350216065836 9 |
|     098517285379256817951564768494112531847337135614230294483 3 |
|     599566496585408242307513495553588841170935156792462642610 6 |
|     325688617701626288802749340655947721494876042230599608848 9 |
|     888565389167675725288023161955083262900265331204553163450 8 |
|     332591772750592260280657863432041248565056032694910686915 3 |
|     743118966082359517504852194508794639986                    ] |
| THE ANSWER FOR 10 FEET IS                                        |
|  [ 500.000040754757403673084082919797630180512327215229061109 7 |
|     650817149143683241973152003319426877820766831310742892112 2 |
|     074122072617284752919175416633910948799809071822219337336 3 |
|     869892092825989956813205454008650603666111640189920354156 9 |
|     299506112022471755341871580832437224401111879635104990804 8 |
|     994305846187282625902519675662657230588191961812718119675 6 |
|     761448718740040860149819224185268862241617529111596560280 7 |
|     227336529038678906541355758315759227691668078794625813507 1 |
|     595556560773068193957563746723927059230                    , |
|    500.000040754757403673084082919797630180512327215229061109 7 |
|     650817149143683241973152003319426877820766831310742892112 2 |
|     074122072617284752919175416633910948799809071822219337336 3 |
|     869892092825989956813205454008650603666111640189920354156 9 |
|     299506112022471755341871580832437224401111879635104990804 8 |
|     994305846187282625902519675662657230588191961812718119675 6 |
|     761448718740040860149819224185268862241617529111596560280 7 |
|     227336529038678906541355758315759227691668078794625813507 1 |
|     595556560773068193957563746723927060090                    ] |
|------------------------------------------------------------------|
| CPU time =       15505.37 seconds;  Page faults =         142    |
|------------------------------------------------------------------|
```

500 Digit Variable Precision Interval

```
| THE ANSWER FOR  1 FOOT IS                                          |
| [ 69.999999992071399767O1,   69.99999999207139976702 ]             |
| THE ANSWER FOR  2 FEET IS                                          |
| [ 69.99999998494462176135,   69.99999998494462176136 ]             |
| THE ANSWER FOR  3 FEET IS                                          |
| [ 69.99999999895284346426,   69.99999999895284346427 ]             |
| THE ANSWER FOR  4 FEET IS                                          |
| [ 70.00000714552716282032,   70.00000714552716282033 ]             |
| THE ANSWER FOR  5 FEET IS                                          |
| [ 70.00110333253882011031,   70.00110333253882011033 ]             |
| THE ANSWER FOR  6 FEET IS                                          |
| [ 70.07240705773198160895,   70.07240705773198160896 ]             |
| THE ANSWER FOR  7 FEET IS                                          |
| [ 72.05416174750445533986,   72.05416174750445534261 ]             |
| THE ANSWER FOR  8 FEET IS                                          |
| [ 95.78048625580866151535,   95.78048625580866151809 ]             |
| THE ANSWER FOR  9 FEET IS                                          |
| [ 219.179288428267184047 40, 219.17928842826718404742 ]            |
| THE ANSWER FOR 10 FEET IS                                          |
| [ 500.00004075475740366990, 500.00004075475740367420 ]             |
|--------------------------------------------------------------------|
| CPU time =  2396.169230 seconds; Page faults =     223             |
|--------------------------------------------------------------------|
```

200 Digit Variable Precision Interval -- Output Optimized

```
| THE ANSWER FOR  1 FOOT IS                                          |
| [ 69.999999992071399767O1,   69.99999999207139976702 ]             |
| THE ANSWER FOR  2 FEET IS                                          |
| [ 69.99999998494462176135,   69.99999998494462176136 ]             |
| THE ANSWER FOR  3 FEET IS                                          |
| [ 69.99999999895284346426,   69.99999999895284346427 ]             |
| THE ANSWER FOR  4 FEET IS                                          |
| [ 70.00000714552716282032,   70.00000714552716282033 ]             |
| THE ANSWER FOR  5 FEET IS                                          |
| [ 70.00110333253882011031,   70.00110333253882011033 ]             |
| THE ANSWER FOR  6 FEET IS                                          |
| [ 70.07240705773198160895,   70.07240705773198160896 ]             |
| THE ANSWER FOR  7 FEET IS                                          |
| [ 72.05416174750445533986,   72.05416174750445534261 ]             |
| THE ANSWER FOR  8 FEET IS                                          |
| [ 95.78048625580866151535,   95.78048625580866151809 ]             |
| THE ANSWER FOR  9 FEET IS                                          |
| [ 219.179288428267184047 40, 219.17928842826718404742 ]            |
| THE ANSWER FOR 10 FEET IS                                          |
| [ 500.00004075475740366990, 500.00004075475740367420 ]             |
|--------------------------------------------------------------------|
| CPU time =  2377.67 seconds;     Page faults =     153             |
|--------------------------------------------------------------------|
```

200 Digit Variable Precision Interval -- Fully Optimized

SOME EXPERIMENTS USING INTERVAL ARITHMETIC*

Eric K. Reuter, John P. Jeter, J. Wayne Anderson
and Bruce D. Shriver

Computer Science Department
University of Southwestern Louisiana
Lafayette, Louisiana 70504

This paper reviews past experiences and discusses future work in the area of interval arithmetic at the University of Southwestern Louisiana (USL). Two versions of interval arithmetic were developed and implemented at USL [8]. An interval data type declaration and the necessary mathematical functions for this data type were added to Fortran via the preprocessor Augment [4,5]. In the first version, the endpoints of the intervals were represented as single precision floating point numbers. In the other version, the endpoints were represented to 56 decimal digits. Production engineering programs were run as benchmarks [8]. The accumulation of computational and algorithmic error could be observed as a widening of the intervals. The benchmarks were also run in normal single and double precision arithmetic. In some instances, the result obtained from a single or double precision calculation was not bounded by the corresponding interval result indicating some problem with the algorithm. The widening of an interval does not necessarily indicate a data sensitivity nor error in an algorithm. However, these large intervals can be used as indicators of possible trouble areas. On the other hand, small intervals can be used as an indicator of no problems. As could be expected, the 56-decimal digit precision interval gave better results in terms of smaller intervals due to the increased amount of precision. The obvious problem with this version is that the amount of overhead required for its execution is high.

* This work has been supported in part by the U. S. Army Corps of Engineers, contract numbers DACA39-76-M-0249 and DACA39-77-M-0106.

1.0 Introduction

The floating point number system used on contemporary computers is an approximation to the real number system. In interval arithmetic, a non-representable real number is approximated by an interval consisting of machine representable endpoints which bound the number. Intervals will be regarded as bounds on an exact but unknown real number. This means that if the interval [a,b] is a computer approximation to the exact result x then $a \leq x \leq b$. To obtain the "best" machine representation of the interval, a must be the greatest lower bound for x and b must be the least upper bound for x. In this way the interval [a,b] will be the smallest computer representable interval that contains x.

In order to obtain the smallest computer representable interval for the result of arithmetic operations on intervals, directed roundings on the computer arithmetic operations must be defined. If x is a real number and M1 and M2 are two consecutive machine representable numbers such that $M1<x<M2$ and if r is a rounding function, then r is downward directed if $r(x) = M1$ and r is upward directed if $r(x) = M2$. M1 and M2 will be the machine representable numbers that are respectively the greatest lower bound and the least upper bound for the real number x. If x is a machine representable number, then $r(x) = x$.

In general, the result of a finite precision arithmetic operation does not always produce a machine representable number. In other words, a op b, where a and b are machine representable

numbers and op is, in general, one of the machine arithmetic operations, may not be a machine representable number and must be rounded.

Since the exponent range of floating point numbers is bounded, exponent overflow and underflow may occur during an arithmetic operation. If underflow occurs, then the true result is between zero and the smallest positive or negative representable number. In the case of underflow, a directed rounding may give a valid bound. In the case of overflow, if rounding away from zero is wanted, then there is no machine representable number which can be used as a correct bound. This is known as an infinity fault.

1.1 Interval Valued Functions

A real-valued function, f, which is defined and continuous on an interval [a,b] can be extended to an interval-valued function, F, of an interval variable, [a,b], by defining

$$F([a,b]) = [c,d] \text{ such that } f(x) \text{ is contained in } [c,d] \text{ for every } x \text{ in } [a,b]$$

where c and d are machine representable numbers.

When f is evaluated at a point x using a machine representable approximation to x, a computer approximation to f results. This computer approximation, F([a,b]), is defined as an interval that contains f(x). If f is monotonic increasing on [a,b], then F([a,b]) = [rd(f(a)),ru(f(b))] where rd is such that rd(f(a)) $\leq$ f(a) and ru is such that ru(f(b)) $\geq$ f(b). Ideally, we

would like rd(f(a)) to be the largest machine representable number such that rd(f(a)) $\leq$ f(a) (i.e., a greatest lower bound) and ru(f(b)) to be the smallest machine representable number such that ru(f(b)) $\geq$ f(b) (i.e. a least upper bound). Similarly, if f is monotonic decreasing on [a,b], then F([a,b]) = [rd(f(b)),ru(f(a))].

If f is not monotonic on [a,b], then the interval [a,b] can be divided into disjoint subintervals; $[a_i, b_i]$, i = 1,2,3,...,n; where each a_i and b_i are machine representable numbers and f is monotonic on each subinterval. Further, $\bigcup_i [a_i, b_i]$ contains all machine representable numbers in the interval [a,b] and f is monotonic on each subinterval. It can be shown in this case that $F([a, b]) = \bigcup_i F([a_i, b_i])$.

Algorithms for performing the machine arithmetic operations with directed roundings can be found in Yohe [9]. These operations are used to compute the endpoints of the resultant interval for a particular arithmetic operation performed on two intervals. A downward directed rounding is performed on the left endpoint and an upward directed rounding is performed on the right endpoint. For example, interval addition is defined as follows:

$$[a,b] + [c,d] = [rd(a \oplus c), ru(b \oplus d)]$$

where $\oplus$ is the machine addition operation and rd is a downward directed rounding and ru is an upward directed rounding.

It may not be possible to obtain the best bounds for the result of the computer approximation to the function f. An example would be a machine calculation of the sine which is known to be accurate to only 7 digits out of 9.

2.0 The Implementation of the MRC Interval Arithmetic Package for the Multics System

The interval arithmetic package and the input/output routines for interval numbers which have been implemented on the Multics system follow the design of an interval arithmetic package implemented on the UNIVAC 1108 computer located at the Mathematics Research Center, MRC, of the University of Wisconsin [2,6,10]. A description of the implementation of the MRC interval arithmetic package on the Multics system is given in Appendix A. This appendix is quite lengthy but contains information related to the implementation of mathematical software rarely found in the literature.

3.0 Benchmarks

Several production programs were obtained from the Army Corps of Engineers, Waterways Experiment Station, Vicksburgh, Mississippi, to be run as benchmarks. These programs consisted of four linear equation solvers, a matrix inversion routine, a fast fourier transform routine, a slope stability program and a stress program.

The accumulation of computational and algorithmic error can be seen as a growth in the width of intervals. Wide intervals

are not necessarily a sign of data sensitivity or algorithmic error. When a program is run using interval data types, a natural tendency is for intervals to grow wider. However, small intervals are an indication of no problems and wide intervals serve as indicators of possible trouble spots.

During the testing of the initial interval implementation, there were many instances where the intervals became quite large. It was difficult to determine during analysis whether this widening was a problem with the algorithm, an unavoidable result from interval arithmetic, or due to the lack of precision of the representation of the endpoints. 56 decimal digit interval was implemented to help resolve this problem.

3.1 Linear Equation Solvers

Four linear equation solvers were included in the benchmarks supplied by the Army Corps of Engineers. Included was a Gaussian elimination program. It was first tested on a simple 4 by 4 linear system. Using the standard interval package, the magnitude of the resulting intervals were from 10**-4 to 10**-2. All routines were also run in regular single and double precision. The results obtained by using standard interval insured the correctness of the results only to the third or fourth decimal place. In all instances the intervals bounded the results produced in single and double precision. The same test case was executed using the 56 decimal digit interval package. In this case the width of the intervals varied from 10**-51 to 10**-50. This extra precision obtained from using extended

precision interval was obtained at the cost of an increase in cpu time used. The standard interval run required only .44 seconds of processor time while the extended interval required 12.64 seconds. More will be said about the cost of interval and extended interval later.

A second test case, this time a 7 by 7 linear system, was also tried. The standard interval version did not produce any results as the intervals grew too large. However, the extended interval version was able to compute results. The width of the intervals produced varied from 10**-45 to 10**-43.

There were three other equation solvers. The second equation solver, BANSOL, solved banded systems of equations using Gaussian elimination with no pivoting. The matrix of coefficients is assumed to be symmetrical and only the upper triangular banded matrix of coefficients is stored. The SESOL program solved a banded system of linear equations using the LU decomposition technique. Operations with zero elements are not performed. The matrix of coefficients is symmetrical and only the upper triangular banded matrix of coefficients is stored. The fourth equation solver was a spline program. It solved a system of linear equations using an iterative technique to calculate the moments of a set of data points in order to fit a cubic spline to those data points. In all three cases the results were similar to those above and are discussed in detail in [?].

3.2 Matrix Inversion

The matrix inversion program finds the inverse of a square

matrix. The first test case was a Hilbert matrix of order 4. The interval results from the standard interval run were quite wide, from 10**-3 to 0.26. The extended intervals were from 10**-50 to 10**-47. When an attempt was made to invert a Hilbert matrix of order 10, standard interval could not find a solution and the single precision results were erroneous. The extended precision intervals widths ranged from 10**-36 to 10**-28 and again indicated that the double precision results were good to only 8 or 9 digits of precision.

3.3 Fast Fourier Transform

The fast fourier transform (FFT) program supplied by the Army Corps of Engineers proved to be a quite stable algorithm. A difficulty in its implementation in double precision and interval should be mentioned. A FFT program produces complex arithmetic results. Fortran does not normally support double precision complex arithmetic and, therefore, it had to be simulated. The same type of simulation had to be done for interval. This slowed the execution of the algorithm considerably. In all test cases, all arithmetics produced good results. The single precision intervals had a width of on the order of 10**-6 and the extended intervals, 10**-53.

3.4 Slope Stability Program

An application program, SLOPE, was also sent us by the Army Corps of Engineers. Testing using this program consisted of varying a set of three inputs (cohesion, unit weight, and phi)

for the program plus or minus ten percent. This resulted in 81 runs for each type of arithmetic.

Two problems arose when implementing the slope program in interval arithmetic. The first resulted from the way in which the interval package evaluates the test value in an arithmetic IF statement. When an arithmetic IF statement is encountered with an interval test value, the interval is converted to real, i.e., the midpoint is taken. In one of the subroutines a particular branch was to be taken only if the test value is positive. Certain intervals were passing along this branch whose midpoint was indeed positive but whose left endpoint was negative. The interval was subsequently used as a divisor and, since the interval contained zero, a zero divide error occured. The solution to this problem was to recode using a logical IF which is evaluated in a different manner and avoids this problem.

The second problem was more difficult to pin down. During testing using standard interval, some of the runs contained intervals which were "blowing up", that is, the width of the intervals were becoming unacceptably large. After a considerable analysis effort, a correlation was uncovered between the large intervals and the -10% value for unit weight. By starting with the initial value for unit weight and decreasing its value in increments of .25%, the initial value at which the intervals blew up could be pinpointed. This occured at about -2.25% of the initial value. As long as unit weight did not go below this value, acceptable results were obtained. After further effort,

the problem was traced to a single statement, "T3 = FS1 - FSL". As unit weight decreased below -2.25% of its original value, values of FS1 and FSL became closer and closer together. This subtraction resulted in stripping off the significant digits. T3 was subsequently used as a divisor compounding the effect.

During the procedure of tracking down the error source, a side benefit was reaped which is indicative of the type of recoding of algorithms sometimes necessary to get satisfactory results from limited precision interval arithmetic. Several computations could be combined and an interval consistently of less than optimal width could be factored out producing a more accurate algorithm. The set of runs was repeated using the extended interval package. Most of the data sensitivity noted above disappeared. No interval widths exceeded 10**-4.

3.5 Testing Summary

The 56-decimal digit interval package did prove useful in many cases. Often the standard interval either produced no solution or solutions with extremely wide intervals. Some massaging of the code supplied by the Army Corp of Engineers was required to execute it satisfactorily using interval arithmetic. The primary cost of the use of extended precision interval arithmetic was in terms of central processing time consumed and increased paging activity. On a system like Multics, both of these figures can be perturbed by the load on the system. The figures in Table 3.1 for the FFT routine indicates a general trend. This data was gathered from runs made during a contiguous

time interval during a period of low system utilization.

	PAGE_FAULTS	CPU_TIME (seconds)
single precision	23	0.3623
double precision	36	0.6678
standard interval	39	14.4994
56 decimal interval	3195	466.8781

Table 3.1
FFT Subroutine Overhead

4.0 Conclusions and Future Work

Interval arithmetic can, at times, be extremely useful. For instance, it can be used to indicate the limits of precision of an algorithm for a given set of data. From the testing it was shown that much better bounds on the precision were obtained using the extended interval package. This was, of course, not unexpected. 56 decimal digits carry more precision than 27 binary digits (equivalent to approximately 8 decimal digits) and there is no conversion error on input and output for the 56 decimal interval package. The price paid was in terms of runtime efficiency. Standard precision interval resulted in approximately, at most, an order of magnitude increase in execution time over that of single or double precision arithmetic. 56 decimal interval arithmetic resulted in a further increase of more than one to more than two orders of magnitude. It should be noted here that the 56 digit version was based upon the 59 decimal digit hardware arithmetic unit of the Honeywell

H68/80 processor. Extended precision arithmetic using software simulated basic operations could be expected to take much longer.

One obvious application of extended interval arithmetic would be to validate existing programs. Any data sensitivity discovered could be included in a description of the algorithm and directions on its use. Although extended precision interval arithmetic is expensive, its cost must be balanced against possible consequences of using invalid results. An organization like Corps of Engineers might weigh a defective dam or the cost of moving 100,000 tons of dirt against the cost of a few hours of computer time.

A more effective technique would be to first test the algorithm using standard precision interval arithmetic. Its relatively small decrease in run time efficiency indicates that its use is more than justified as an economical means of identifying possible trouble areas in an algorithm for the data under consideration. The more expensive extended interval package could be applied to just those cases where possible trouble areas have been identified.

Interval arithmetic can be used to determine the precision of the arithmetic required to guarantee a given precision in the results of an algorithm. In some of the benchmarks executed in 56 decimal digit interval arithmetic, the results were good only to 40 or so digits. This represents a considerable loss of precision. It also points out why arbitrarily picking a given precision for arithmetic does not guarantee results in which

absolute confidence can be placed. How great an increase in precision is obtained, if any, by going from a machine with 32-bit words to one with 60 bit words?

In general, whether using interval or regular arithmetic, the greater the precision the longer the run time required for a given algorithm. Having variable precision interval arithmetic would allow the validation of algorithms for which standard precision interval arithmetic is insufficient without having to go all the way to 56 decimal digit precision. There will also be instances where it might be desirable or necessary to go beyond 56 decimal digits of precision. In any case, the overhead associated with execution in interval arithmetic will only be as great as required for the necessary precision. A variable precision interval arithmetic package is currently under development at USL.

The execution speed of interval arithmetic can be increased in several ways. One would be to decrease the number of levels of interpretation required in the current implementation. The optimum solution would be to have a hardware or firmware module which could execute variable precision interval arithmetic. Many existing minicomputer systems have undefined opcodes for just such requirements. As a side effect, an arithmetic unit that can execute variable precision interval arithmetic can also execute traditional variable precision floating point arithmetic. This means that interval arithmetic, of the necessary precision, could be used to determine the required arithmetic precision

required for the results of the algorithm. The algorithm, then, could be executed using only the required precision.

Appendix A

A.0 The Implementation of the MRC Interval Arithmetic Package for the Multics System

In the Multics implementation, the endpoints of the intervals are represented as a pair of floating point numbers stored in consecutive storage locations. The Multics single precision floating point format uses a 36 bit word which consists of an 8-bit 2's complement exponent, with the high order bit the sign bit, followed by a 28-bit normalized 2's complement fraction, with the high order bit the sign bit.

The subroutines of the MRC interval package can be divided into eight categories. These categories are arithmetic operations, exponentiation operations, conversion functions, comparison, basic external functions, supporting functions, input/output routines and miscellaneous. All of the routines in each category except the input/output category were written in Fortran. Several of the Fortran subroutines call routines that are written in PL/I. The PL/I routines correspond for the most part to the assembler routines that were written for the UNIVAC 1108 version of the interval package and are written specifically for the Multics implementation. Most of the input/output routines were written in PL/I.

The routines which perform the four basic arithmetic operations of addition, subtraction, multiplication, and division on interval numbers are machine dependent. Since we want the

best computer approximation to the results of computer arithmetic operations on intervals, directed roundings on the computer arithmetic operations must be performed. The floating point hardware on the system does not perform directed roundings. Therefore the four basic single precision floating point computer arithmetic operations of addition, subtraction, multiplication, and division had to be simulated in order to provide the correct roundings. A description of the routines that simulated the floating point computer arithmetic operations and provided the proper directed roundings and a description of the routines that perform the basic computer arithmetic operations on intervals follows. These routines perform the "best possible arithmetic" computer operations with directed roundings as described by Yohe [9]. All the routines are written in PL/I for the Multics system.

A.1 Basic External Functions

Included in the interval package are the interval counterparts of the Multics basic external functions atan2, exp, alog, alog10, sin, cos, tan, asin, acos, atan, sinh, cosh and sqrt. The general method of calculation of the interval functions involves bounding the results of the corresponding double precision basic external function. For functions that are monotonic over an interval, the endpoints of the resultant interval are the result of the double precision function evaluated at the endpoints of the input interval and then

properly bounded. If the function is not monotonic over the interval, then a case analysis is done by dividing the input interval into subintervals over which the function is monotonic.

The result obtained from the double precision functions must be bounded before it can be used as the endpoint of an interval. Therefore, the accuracy of the results of the double precision basic external functions are required by determining a lower bound on the number of bits of the fraction that the result is guaranteed to have. This can be illustrated by the following example. Suppose a result is accurate to 35 bits of fraction and a 27 bit lower bound for the result is required. Assume that the 27th through 37th bits of the fraction were 10000000000. If the result were just truncated to 27 bits the 27th bit would be a 1. If however the 37th bit was one unit too large, then bits 27 through 37 would be 01111111111 and the 27th bit of the correct lower bound would be 0. It cannot be determined which case is correct.

The following general bounding technique is performed which will produce correct bounds in all cases, but it does not necessarily produce optimal bounds. If a lower bound is sought for the double precision result, then the fraction is decremented by one at or before the last bit known to be accurate. If an upper bound is sought, then the fraction is incremented by one at or before the last bit known to be accurate. The same bounding technique used in bounding the results of the arithmetic

operations is then used to obtain the 27 bit fraction of the result.

A.1.1 Accuracy Testing

To our knowledge, there is no documentation concerning the implementation of the basic external functions on Multics used by PL/I and Fortran. Therefore, the accuracy of these functions had to be determined. Three approaches were considered for use in determining the accuracy of the required external function:

1) rigorous error analysis of actual implementations
2) rewriting of the algorithms
3) comparison of accuracy with known test data

First, the error analysis of the mathematical library routines seemed to be impossible due to the: (a) lack of description of the algorithms employed, (b) low readability of the source programs (much of which was written in ALM, the assembly language of Multics). The second possibility had to be eliminated due to the time constraints of the project and therefore the third approach had to be taken.

The testing itself was done in two stages:

stage1 - generation of input test data and evaluation of the given function

stage2 - comparison of significant digits of the result and corresponding values in published tables [1]

"Driver" programs were written which generated test data and called the routines which were to be tested. The output was generated in decimal form and then a check was made as to the first digit that was different from the result given in the table. All digits of function values which were tested proved to be identical with corresponding tabular digits (the only exception being the last digit in the Abramovitz's tables). However, the analysis of the very next digit in our results showed that in each case the error was caused by an upward rounding.

The test data had been restricted to the decimal values that can be represented exactly in the floating binary notation. Thus, we avoided the input conversion error and the function value could be obtained for the true argument. Also, we have to warn that the accuracy estimated in this way must be somewhat pessimistic. We were able to check only as many digits as were given in the standard tables. Thus, the tan function is assumed to have only 8 accurate decimal digits even though there are reasons to believe that accuracy is much greater than that.

2.1.2 Error Conditions

The Univac 1108 double precision floating point number has an 11-bit exponent field vs. an 8-bit exponent in the single precision word. This allowed the checking for overflow and underflow faults to be done during the conversion from double to

single precision format. In Multics, both single precision and double precision floating point numbers have an 8-bit exponent field. Therefore, the check for eventual fault conditions had to be made prior to the calls to the double precision functions.

Overflow could be produced by the following functions: exp, sinh, cosh and tanh. In the Multics implementation of interval arithmetic, overflow in these functions was prohibited by restricting the domain of the arguments to the interval [-88.028,88.028]. Should an argument fall outside this domain, special actions (described later) had to be taken. Restricting arguments to this domain prevented overflow from occuring during the evaluation of the functions. However, the magnitude of the endpoints of the results were always much smaller than the largest representable number. This implies that the domain of the arguments should be extended.

A.2 Input/Output Routines

The I/O routines implemented on Multics were designed to some extent after the I/O routines implemented for the UNIVAC 1108 version of the interval package [4]. Additional routines were included in the Multics version to handle scalar interval variables and a matrix of interval variables.

A.3 56 Decimal Digit Interval Implementation

A 56 decimal digit version of the original Multics interval package has also been implemented on the Multics system. This version uses the decimal arithmetic hardware available on the

Honeywell H68/80 processor. The decimal arithmetic unit performs both fixed and floating point 59-decimal digit arithmetic. Fixed decimal arithmetic was used to implement the decimal interval package. Floating point decimal arithmetic was not used due to the lack of control the user has over both the rounding strategy used and the detection of faults (overflow, underflow, and divide by zero). The endpoints of the intervals are represented by a 56 decimal digit fraction and a 17 binary digit exponent. A 59-decimal digit fraction was not used because in the implementation of the BPA routines, two digits were needed for guard digits and one digit was reserved for overflow.

The implementation of the 56 decimal digit interval package followed the implementation of the original Multics interval package as closely as possible. In this way the logic of the original interval package was used and the number of errors encountered in the implementation could be reduced. The entire 56 decimal digit interval package was written in PL/I as Fortran does not support decimal arithmetic. Only the number of words required to carry the PL/I representation of the interval was declared in Fortran. The Fortran routines would carry the interval to be passed to the PL/I routines.

The first step in the implementation of the 56 decimal digit interval package was the implementation of the best possible arithmetic , BPA, routines (see section 1.0 of the attached paper). The existing proceedures for doing BPA for the original

interval package were modified to perform 56 decimal version. In the single precision interval package the implementation of the I/O routines proved to be one of the most difficult tasks. This was due to the required conversions between floating decimal and floating binary. The correct roundings had to be done for the conversions in either direction and the algorithms for the conversions became rather involved. The implementation of the 56 decimal digit interval I/O presented no such problems as the internal representation of the interval was already in decimal. The only rounding done is on output when the user requests less than 56 decimal digits of precision.

In the initial interval effort, the interval counterparts of the basic external functions were implemented through the double precision floating binary routines in the Multics library. This obviously would not be sufficient for the 56 decimal implementation. The basic external functions had to be calculated to a precision of greater than 56 decimal digits. To achieve this, the Fortran Multiple Precision Package, MPP, developed by Brent [3] was used. The values produced by the basic external functions could be calculated to an arbitrary precision using MPP. It was necessary to construct an interface between Brent's routines written in Fortran and the interval package written in PL/I. The implementation of the SIN and COS routines presented an especially difficult implementation problem. The arguments had to be reduced to a value between 0 and 2pi. A case analysis then had to be made for each endpoint

to determine the correct interval evaluation of the SIN or COS function. The case analysis depended on the correct 56 decimal digit bounds on the numbers pi/2, pi, 3pi/2, 2pi, 5pi/2, 3pi and 7pi/2. These constants had to computed using the MPP.

References

[1] Abramovitz, M. and Stegun, I. A., (ed.), Handbook of Mathematical Functions, National Bureau of Standard Applied Mathematics Series, June, 1964.

[2] Binstock, W., Hawkes, J. and Hsu, N., "An interval input/output package for the UNIVAC 1108," The University of Wisconsin, Mathematics Research Center, Technical Summary Report No. 1212, September, 1973.

[3] Brent, R. P., "A fortran multiple-precision arithmetic package," Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, May, 1976.

[4] Crary, F. D., "The AUGMENT precompiler, I. User information," The University of Wisconsin, Mathematics Research Center, Technical Summary Report No. 1469, December, 1974.

[5] Crary, F. D., "The AUGMENT precompiler, II. Technical documentation," The University of Wisconsin, Mathematics Research Center, Technical Summary Report No. 1470, October, 1975.

[6] Ladner, T. D. and Yohe, J. M., "An interval arithmetic package for the UNIVAC 1108," The University of Wisconsin, Mathematics Research Center, Technical Summary Report No. 1055, May, 1970.

[7] Moore, R. E, Interval Analysis, Prentice-Hall Inc., Englewood Cliffs, N. J., 1966.

[8] Reuter, E. K. and Podlaska-Lando, S., "Source Listing for the MULTICS Interval Arithmetic Package," Computer Science Department Report No. 76-7-3, University of Southwestern Louisiana, Lafayette, Louisiana, August, 1976.

[9] Yohe, J. M., "Best possible floating point arithmetic," The University of Wisconsin, Mathematics Research Center, Technical Summary Report No. 1054, March, 1970.

[10] Yohe, J. M., "Software for interval arithmetic: a

reasonably portable package," *Transactions on Mathematical Software*, to be published.

Report No. 77-7-1
1977

A 56 Decimal Digit Implementation of an Interval Arithmetic Package on the MULTICS System**

Computer Science Department
University of Southwestern Louisiana
Lafayette, Louisiana 70504

by

Eric Reuter, J.P.Jeter, J.W.Anderson, B.D.Shriver

** This work was supported under contract DACA39-76-M-0249 from Department of the Army, Computer Analysis Branch, Waterways Experiment Station, Corps of Engineers.

56_Decimal_Digit_Interval_FORTRAN_Work

A 56 decimal digit version of the original Multics interva package has been implemented on the Multics system. This versio uses the decimal arithmetic hardware available on the Multic system. The Multics decimal arithmetic unit performs both fixe and floating point 59 decimal digit arithmetic. Fixed decima arithmetic was used to implement the decimal interval package The floating point decimal arithmetic was not used because of th lack of control the user has over both the rounding strategy use and the detection of faults (overflow, underflow and divide b zero). The end points of the intervals are represented by 5 decimal digit fraction and a 17 binary digit exponent. A 5 decimal digit fraction was not used was because in th implementation of the "best possible arithmetic" routines, tw digits were needed for guard digits and one digit was reserve for overflow.

The implementation of the 56 decimal digit interval packag followed the implementation of the original Multics interva package as closely as possible [3]. In this way the logic of th original interval package was used and the number of error encountered in the implementation of the 56 decimal digi interval package was reduced. The entire 56 decimal digi interval package was written in PL/I as Fortran does not suppor decimal arithmetic. Just the number of words required to carr the PL/I representation of the interval was declared in Fortran. The Fortran routines would carry the interval to be passed to th PL/I routines.

The first step in the implementation of the 56 decimal digi interval package was the implementation of the "Best Possibl Arithmetic" or "BPA" routines as proposed by Yohe [4]. Th already existing procedures for doing BPA for the origina interval package were modified to perform 56 decimal "BPA". Th implementation was fairly straightforward.

In the single precision interval package supplied by MRC [2] th implementation of the I/O package proved to be one of the mos difficult operations. The reason was because of the conversio from floating decimal to floating binary and vice versa. Th correct roundings had to be done for the conversions in either direction and the algorithms for the conversions became rathe involved. The implementation of the 56 decimal digit interva I/O presented no problem with conversion since the interna representation of the interval was already in decimal. The onl rounding is on output if the user requests less than 56 decima digits on output.

In the initial interval effort [3] the interval counterparts o the basic external functions were implemented through the doubl precision floating binary routines in the Multics library. Thi

obviously would not be sufficient for the 56 decimal implementation. The basic external functions had to be calculated to a precision of greater than 56 decimal digits. To achieve this, we are using the Fortran Multiple Precision Package, MPP, developed by Richard Brent [1]. The basic external functions could be calculated using the MPP to an arbitrary number of decimal digits. One problem that was to be solved was the interface between Brent's routines written in Fortran and the 56 decimal digit interval package written in PL/I. The interface was just a conversion from the data representation in the MPP to the data representation used in the interval package after a correct rounding was made. Another problem was in the implementation of the SIN and COS routines. These routines required the greatest amount of work to implement. The argument had to be reduced to between 0 and 2pi. A case analysis then had to be made for each endpoint to determine the correct interval evaluation of the SIN or COS function. The case analysis depended on the correct 56 decimal digit bounds on the numbers pi/2, pi, 3pi/2, 2pi, 5pi/2, 3pi and 7pi/2. These constants had to be computed using the MPP.

Summary_Tables

Gaussian_Elimination

An example is shown below of a simple 4 by 4 linear system. The results for single precision, double precision, single precision interval, and extended interval are shown. For the interval results, the width of the intervals appear below each interval. The widths for the single precision intervals are expressed as a single precision value and the widths for the extended intervals are expressed as an extended interval. The widths of the single precision intervals are of the magnitude 10**-4 to 10**-2 while the widths of the extended intervals are of the magnitude 10**-51 to 10**-50. The reduced interval widths for the extended intervals is due to the extra precision of the extended intervals. The price that had to be paid for the increase in precision was an increase in cpu time from .44 cpu seconds to 12.64 cpu seconds for the extended interval results, but a good deal of precision was gained.

Matrix of Order 4:

5	7	6	5
7	10	8	7
6	8	10	9
5	7	9	10

Vector of Constants:

23 32 33 31

Single_Precision

TIME AND PAGE FAULTS FOR THE GAUSSIAN ELIMINATION ARE AS FOLLOWS:

CPU time = 0.011512 seconds; Page faults = 1

THE SOLUTION IS AS FOLLOWS:

0.999999449	1.000000328
1.000000179	0.999999903

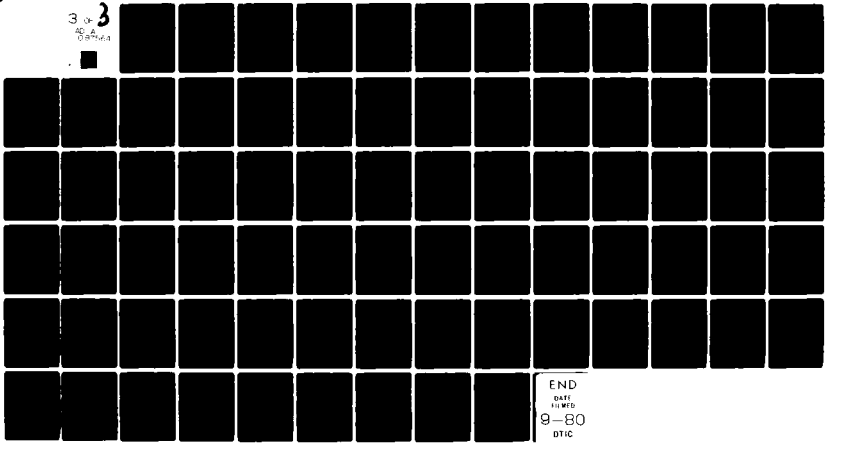
Double_Precision

TIME AND PAGE FAULTS FOR THE GAUSSIAN ELIMINATION ARE AS FOLLOWS:

CPU time = 0.032315 seconds; Page faults = 7

THE SOLUTION IS AS FOLLOWS:

AD-A087 564 UNIVERSITY OF SOUTHWESTERN LOUISIANA LAFAYETTE F/G 9/2
VARIABLE PRECISION AND INTERVAL ARITHMETIC: PORTABLE ENHANCEMEN--ETC(U)
JUL 80 B D SHRIVER DAAG29-78-G-0068
UNCLASSIFIED ARO-15169.1-M NL



```
1.00000000000000003      0.9999999999999998
0.99999999999999999      1.9999999999999999
```

Single_Precision_Interval

```
TIME  AND PAGE FAULTS FOR THE GAUSSIAN ELIMINATION AR AS FOLLOW

CPU time =     0.444765 seconds; Page faults =            3

THE SOLUTION IS AS FOLLOWS:

   [ .99886015+00, .100114449+01]   [ .99929769+00, .1000066993
         0.1142337918e-02                  0.70080116109e-03

   [ .99991752+00, .10000813+001]   [ .99995276+00, .10000480+0
         0.8184090257e-04                  00.4761666059e-04
```

Extended_Interval

```
TIME AND PAGE FAULTS FOR THE GAUSSIAN ELIMINATION ARE AS FOLLOW

CPU time =    12.645348 seconds; Page faults =         5

THE SOLUTION IS AS FOLLOWS:

      [ .9999999999+00000, .1000000001+00001]
      [ .2646500000+00050,.26465000000-00050]

      [ .9999999999+00000, .1000000001+00001]
      [ .1623500000-00050, .1623500000-00050]

      [ .9999999999+00000, .1000000001+00001]
      [ .1895800000-00051, .1895800000-00051]

      [ .9999999999+00000, .1000000001+00001]
      [ .1101000000+00051, .1101000000-00051]
```

Another example, shown below, of a 7 by 7 linear system show
that the single precision interval version has broken down, bu
the extended interval version was able to compute the results.
The interval widths were of the magnitude 10**-45 to 100**-43.

Matrix of Order 7:

```
    180180    120120    90090    72072    60060    51480    45045
    120120    90090     72072    60060    51480    45045    40040
    90090     72072     60060    51480    45045    40040    36036
    72072     60060     51480    45045    40040    36036    32760
    60060     51480     45045    40040    36036    32760    30300
    51480     45045     40040    36036    32760    30030    27720
```

```
    45045    40040    36036    32760    30030    27720    25740

Vector of Constants:

                          1    1    1    1    1    1    1

TIME AND PAGE FAULTS FOR THE GAUSSIAN ELIMINATION ARE AS FOLLOWS:

CPU time =      0.19547 seconds; Page faults =          0

THE SOLUTION IS AS FOLLOWS:

            0.000106439        -0.003075291          0.26859137
           -0.101941098         0.188414240         -0.166749334
            0.056536387
```

Double_Precision

```
TIME AND PAGE FAULTS FOR THE GAUSSIAN ELIMINATION ARE AS FOLLOWS:

CPU time =      0.023362 seconds; Page faults =          1

THE SOLUTION IS AS FOLLOWS:

        0.00015540015540038       -0.00419580419580964
        0.03496503496507612       -0.12820512820526593
        0.23076923076945858       -0.20000000000018252
        0.06666666666672321
```

Single_Precision_Interval

```
THE  GAUSS  ELIMINATION  PROCESS HAS BROKEN DOWN BECAUSE NO PIVOT
GREATER THAN THE INPUT TOLERANCE COULD BE FOUND FOR THE 6TH STEP.
```

Extended_Interval

```
TIME AND PAGE FAULTS FOR THE GAUSSIAN ELIMINATION ARE AS FOLLOWS:

CPU time =      25.043362 seconds; Page faults =         11

THE SOLUTION IS AS FOLLOWS:

        [ .1554001554-00003, .1554001555-00003]
        [ .3468206177-00043, .3468206178-00043]


        [-.4195804196-00002,-.4195804195-00002]
        [ .3357609101-00043, .3357609102-00043]
```

```
[ .3496503496-00001, .3496503497-00001]
[ .1802686229-00043, .1802686230-00043]

[-.1282051283+00000,-.1282051282+00000]
[ .6297511335-00044, .6297511336-00044]

[ .2307692307+00000, .2307692308+00000]
[ .1835758441-00044, .1835758442-00044]

[-.2000000001+00000,-.1999999999+00000]
[ .4241149309-00045, .4241149309-00045]

[ .6666666666-00001, .6666666667-00001]
[ .1298809545-00045, .1298809546-00045]
```

BANSOL_PROGRAM

The following example shows the results for the BANSOL routine which solves a banded system of equations using Gaussian elimination with no pivoting. The matrix of coefficients is assumed to be symmetrical and only the upper triangular banded matrix of order 4 as the matrix of coefficients. The single precision interval results have an interval width of the magnitude 10**-1 while the extended interval results have an interval width of the magnitude 10**-48.

Hilbert Matrix of Order 4

Vector of Constants:

1 0 0 0

Single_Precision

TIME AND PAGE FAULTS FOR BANSOL METHOD ARE AS FOLLOWS:

CPU time = 0.029758 seconds; Page faults = 1

THE SOLUTION IS AS FOLLOWS:

```
0.1600010896e+02          -0.1200011921e+03
0.2400028324e+03          -0.1400018272e+03
```

Double_Precision

TIME AND PAGE FAULTS FOR BANSOL METHOD ARE AS FOLLOWS:

CPU time = 0.026642 seconds; Page faults = 1

THE SOLUTION IS AS FOLLOWS:

```
0.160000000000000562d+02    -0.120000000000000581d+03
0.240000000000000001347d-03  -0.140000000000000000855d+03
```

Single_Precision_Interval

```
TIME AND PAGE FAULTS FOR BANSOL METHOD ARE AS FOLLOWS:
    CPU time =      0.484731 seconds; Page faults =          3

THE SOLUTION IS AS FOLLOWS:

 [ .15969114+02, .16030875+02]   [-.12003797+03,-.11996195+03]
        0.3088021278e-01                0.3800058365e-01

 [ .23997602+03, .24002383+03]   [-.14001561+03,-.13998431+03]
        0.2390098572e-01                0.1564788818e-01
```

Extended_Interval

```
TIME AND PAGE FAULTS FOR BANSOL METHOD ARE AS FOLLOW:

CPU time =        1.959542 seconds; Page faults =          4

THE SOLUTION IS AS FOLLOWS:

        [ .1599999999+00002, .1600000001+00002]
        [ .2227200000-00048, .2227200000-00048]

        [-.1200000001+00003,-.1199999999+00003]
        [ .2740700000-00048, .2740700000-00048]

        [ .2399999999+00003, .2400000001+00003]
        [ .1725100000-00048, .1725100000-00048]

        [-.1400000001+00003,-.1399999999+00003]
        [ .1127100000+00048, .1127100000-00048]
```

The next example uses an Hilbert matrix of order 10 for the matrix of coefficients. The solution for the single precision interval case cannot be found. The solution for the extended interval case could be found with the interval widths ranging from 10**-29 to 10**-25. Also note that the single precision results are meaningless.

```
Hilbert Matrix of Order 10

Vector of Constants:

                    1  0  0  0  0  0  0  0  0  0
```

Single_Precision

TIME AND PAGE FAULTS FOR BANSOL METHOD ARE AS FOLLOWS:

CPU time = 0.065116 seconds; Page faults = 1

0.6804037952e+02	-0.2041981995e+04	0.1802587769e+05
-0.6609998340e+05	0.1038329834e+06	-0.3899778605e+05
-0.5230199170e+05	0.9572640381e+04	0.6173250244e+05
-0.3380692432e+05		

Double Precision

TIME AND PAGE FAULTS FOR BANSOL METHOD ARE AS FOLLOWS:

CPU time = 0.099313 seconds; Page faults = 1

THE SOLUTION IS AS FOLLOWS:

0.1000000023578835879 3d+03	-0.4950000202446120795 2d+0
0.7920000429216765519 4d+05	-0.6006000388819666227 4d+0
0.2522520184935564124 3d+07	-0.6306300507208783912 6d+0
0.9609600830559078902 3d+07	-0.8751600801311593106 2d+0
0.4375800420080998475 0d+07	-0.9237800922669556988 5d+0

Single Precision Interval

SOLUTION CANNOT BE FOUND

Extended Interval

TIME AND PAGE FAULT FOR BANSOL METHOD ARE FOLLOWS:

CPU time = 13.190437 seconds; Page faults = 16

THE SOLUTION IS AS FOLLOWS:

[.9999999999+00002, .1000000001+00003]
[.2607111740-00025, .2607111741-00025]

[-.4950000001+00004,-.4949999999+00004]
[.3216633306-00025, .3216633307-00025]

[.7919999999+00005, .7920000001+00005]
[.2078321976-00025, .2078321977-00025]

[.6006000001+00006,-.6005999999+00006]
[.9134022350-00026, .9134022351-00026]

[.2522519999+00007, .2522520001+00007]
[.3044631515-0026, .3044631516-00026]

[-.6306300001+00007,-.6306299999+00007]
[.8177182762-00027, .8177182763-00027]

[.9609599999+00007, .9609600001+00007]
[.1839461895-00027, .1839461896-00027]

[-.8751600001+00007,-.8751599999+00007]
[.3552142351-00028, .3552142352-00028]

[.4375799999+00007, .4375800001+00007]
[.5810584606-00029, .5810584607-00029]

[-.9237800001+00006,-.9237799999+00006]
[.1288532381-00029, .1288532382-00029]

SESOL_Program

The SESOL program solves a banded system of linear equations using the LU decomposition technique. Operations with zero elements are not performed. The matrix of coefficients is symmetrical randomly; only the upper triangular branded matrix of coefficients is stored. The first example uses an Hilbert matrix of coefficients is stored. The first example uses an Hilbert matrix of order 4 for the matrix of coefficients. The single precision interval results had interval widths of the magnitude 10**-1, while the extended interval results had interal widths of the magnitude 10**-48.

Hilbert Matrix of Order 4

Vector of Constants:

1 0 0 0

Single_Precision

TIME AND PAGE FAULTS FOR SESOL METHOD ARE AS FOLLOWS:

CPU time = 0.223950 seconds; Page faults = 5

THE SOLUTION IS AS FOLLOWS:

0.1600008774e+02	-0.1200009499e+03
0.2400022411e+03	-0.1400014400e+03

Double_Precision

TIME AND Page Faults TIME AND PAGE FAULTS FOR SESOL METHOD ARE AS FOLLOWS:

```
CPU time =      0.225749 seconds; Page faults =      6

THE SOLUTION IS AS FOLLOWS:

   0.15999999999999999542d+02    -0.11999999999999999407d+03
   0.23999999999999998501d+03    -0.13999999999999998998d+03
```

Single_Precision_Interval

```
TIME AND PAGE FAULTS FOR SESOL METHOD ARE AS FOLLOWS:

CPU time =      0.657901 seconds; Page faults =       9

THE SOLUTION IS AS FOLLOWS:

   [ .1596054+02, .16033943+02]    [-.12004178+03,-.11995823+03]
         0.3394412994e-01                0.4177045822e-01

   [ .23997377+03, .24002631+03]   [-.14001725+03,-.13998282+03]
         0.262670570e-01                 0.1720905304e-01
```

Extended_Interval

```
TIME AND PAGE FAULTS FOR SESOL METHOD ARE AS FOLLOWS:

CPU time =      2.108744 seconds; Page faults =       49

THE SOLUTION IS AS FOLLOWS:

       [ .1599999999+00002, .1600000001+000002]
       [ .2479670000-00048, .2479670000-000481

       [-.1200000001+00003,-.1199999999+00003]
       [ .3051400000-00048, .3051400000-000481

       [ .2399999999+00003, .2400000001+00003]
       [ .1920100000-00048, .1920100000-000481

       [-.1400000001+00003,-.1399999999+00003]
       [ .1255600000-00048, .1255600000-000481
```

The next example uses an Hilbert matrix of order 10 for the matrix of coefficients. The single precision and single precision interval cases could not find a solution. The extended interval results had interval widths of the magnitude 10**-29 to 10**-25.

Hilbert Matrix of Order 10

Vector of Constants:

1 0 0 0 0 0 0 0 0 0

Single_Precision

STOP *** ZERO DIAGONAL ENCOUNTERED DURING EQUATION SOLUTION
EQUATION NUMBER = 9

Double_Precision

TIME AND PAGE FAULTS FOR SESOL METHOD ARE AS FOLLOWS:

CPU time = 0.243050 seconds; Page faults = 6

THE SOLUTION IS AS FOLLOWS:

0.10000000071884660358d+03	-0.49500000625171377493d+04
0.79200001337832115041d+05	-0.60060001220562858816d+06
0.25225200583827033424d+07	-0.63063001608613853932d+07
0.96096002644256749918d+07	-0.87516002559459038221d+07
0.43758001345532895439d+07	-0.92378002962530892762d+06

Single_Precision_Interval

STOP *** ZERO DIAGONAL ENCOUNTERED DURING EQUATION SOLUTION
EQUATION NUMBER = 6

Extended_Interval

TIME AND PAGE FAULTS FOR SESOL METHOD ARE AS FOLLOWS:

CPU time = 12.364521 seconds; Page faults = 93

THE SOLUTION IS AS FOLLOWS:

[.9999999999+00002, .1000000001+00003]
[.2961612465-00025, .2961612466-00025]

[-.4950000001+00004,-.4949999999+00004]
[.3654013424-00025, .3654013425-00025]

[.7919999999+00005, .7920000000+00005]
[.2360920776-00025, .2360920777-00025]

[-.6006000001+00006,-.6005999999+00006]
[.1037601650-00025, .1037601651-00025]

[.2522519999+00007, .2522520001+00007]
[.3458623775-00026, .3458623776-00026]

[-.6306300001+00007,-.6306299999+00007]
[.9289071131-00027, .9289071132-00027]

[.9609599999+00007,-.8751599999+00007]
[.2089581813-00027, .2089581814-00027]

[-.8751600001+00007,-.8751599999+00007]
[.4035143144-00028, .4035143145-00028]

[.4375799999+00007, .4375800001+00007]
[.6600675959-00029, .6600675960-00029]

[-.9237800001+00006,-.9237799999+00006]
[.1463739935-00029, .1463739936+00029]

SPLINE

The spline program solves a system of linear equations using a iterative technique to calculate the moments at a set of data points in order to fit a cubic spline to these data points. The first example uses 4 (X,Y) data points. The single precision interval widths were of the magnitude 10**-5. The extended interval widths were of the magnitude 10**-53.

(X,Y) DATA POINTS:

X	Y
1.6	1
5.4	2
7	1
8.2	1

Single_Precision

TIME AND PAGE FAULTS FOR SPLINE ARE AS FOLLOWS:

CPU time = 0.003969 seconds; Page faults = 1

INTERPOLATED VALUES

X	Y
0.1000000000e+01	0.6069527492e+00
0.3000000000e+01	0.1842634425e+01
0.5000000000e+01	0.2160504758e+01
0.7000000000e+01	0.1000000000+01
0.9000000000e+01	0.1135431752e+01

Double_Precision

```
TIME AND PAGE FAULTS FOR SPLINE ARE AS FOLLOWS:

CPU time =      0.007115 seconds; Page faults =      1

INTERPOLATED VALUES

              X                                   Y
   0.10000000000000000000d+01          0.60695274774605015625d+00
   0.30000000000000000000d+01          0.18426344355119746667d+01
   0.50000000000000000000d+01          0.21605047819613091100d+01
   0.70000000000000000000d+01          0.10000000000000000000d+01
   0.90000000000000000000d+01          0.11354317730190776109d+01
```

Single_Precision_Interval

```
TIME AND PAGE FAULTS FOR SPLINE ARE AS FOLLOWS:

CPU time =      0.800636 seconds; Page faults =      4

INTERPOLATED VALUES

X = [ .1000000000000+01, .1000000000000+01]
                    0.0000000000e+00
Y = [ .6069516241550+00,.6069538742330+00]
                    0.1125037670e-05

X = [ .3000000000000+01, .3000000000000+01]
                    0.0000000000e+00
Y = [ .1842632219195+01, .1842636644841+01]
                    0.2212822437e-05

X = [ .5000000000000+01, .5000000000000+01]
                    0.0000000000e+00
Y = [ .2160503506660+01, .2160506069661+01]
                    0.1281499863e-05

X = [ .7000000000000+01, .7000000000000+01]
                    0.0000000000e+00
Y = [ .1000000000000+01, .1000000000000+01]
                    0.0000000000e+00

X = [ .9000000000000+01, .9000000000000+01]
                    0.0000000000e+00
Y = [ .1135428726673+01, .1135434836150+01]
                    0.3054738045e-05
```

Extended_Interval

```
TIME AND PAGE FAULTS FOR SPLINE ARE AS FOLLOWS:

CPU time =      5.429209 seconds; Page faults =       0

INTERPOLATED VALUES

X = [ .1000000000+00001, .1000000000+00001]
    [ .0000000000+00000, .0000000000+00000]
Y = [ .6069527477+00000, .6069527478+00000]
    [ .1775000000-00053, .1750000000-00053]

X = [ .3000000000+00001, .3000000000+00001]
    [ .0000000000+00000, .0000000000+00000]
Y = [ .1842634435+00001, .1842634436+00001]
    [ .3700000000-00053, .3700000000-00053]

X = [ .5000000000+00001, .5000000000+00001]
    [ .0000000000+00000, .0000000000+00000]
Y = [ .2160504781+00001, .2160504782+00001]
    [ .2200000000-00053, .2200000000-00053]

X = [ .7000000000+00001, .7000000000+00001]
    [ .0000000000+00000, .0000000000+00000]
Y = [ .1000000000+00001, .1000000000+00001]
    [ .0000000000+00000, .0000000000+00000]

X = [ .9000000000+00001, .9000000000+00001]
    [ .0000000000+00000, .0000000000+00000]
Y = [ .1135431773+00001, .1135431774+00001]
    [ .3400000000-00053, .3400000000-00053]
```

The next example uses 11 (X,Y) data points. The single precision interval version of the program could not find a solution, while the extended interval version could find a solution with interval widths of the magnitude 10**-51 to 10**-50. (X,Y) DATA POINTS:

X	Y
1.0	1.008
10	20.183
19	339.096
28	58.69
37	85.48
46	106.7
55	132.91
64	156.9
73	180.88
82	207.21
91	231

Single Precision

```
TIME AND PAGE FAULTS FOR SPLINE ARE AS FOLLOWS:
```

```
CPU time =       0.009885 seconds; Page faults =      1

INTERPOLATED VALUES

                X                       Y
  0.1000000000e+01        0.1008000001e+01
  0.5000000000e+01        0.9510347366e+01
  0.1000000000e+02        0.2018300009e+02
  0.1500000000e+02        0.3085654688e+02
  0.2000000000e+02        0.4108214760e+02
  0.2500000000e+02        0.5135257101e+02
```

Double_Precision

```
TIME AND PAGE FAULTS FOR SPLINE ARE AS FOLLOWS:

CPU time =       0.033277 seconds; Page faults =        2

INTERPOLATED VALUES

                X                                       Y
 0.10000000000000000000d+01        0.10079999999999999999d+01
 0.50000000000000000000d+01        0.95103474691975531663d+01
 0.10000000000000000000d+01        0.20182999999999999999d+01
 0.15000000000000000000d+02        0.41082147846906934087d+02
 0.20000000000000000000d+02        0.41082147846906934087d+02
 0.25000000000000000000d+02        0.51352571259161417030d+02
```

Single_Precision_Interval

```
SINGLE  PRECISION  INTERVAL  HAS BROKEN DOWN DUE TO BOUNDS FAULTS
(see attachment)
```

Extended_Interval

```
TIME AND PAGE FAULTS FOR SPLINE ARE AS FOLLOWS:

CPU time =       31.215525 seconds; Page faults =        5

INTERPOLATED VALUES

X = [ .1000000000+00001, .1000000000+00001]
    [ .0000000000+00000, .0000000000+00000]
Y = [ .1008000000+00001, .1008000000+00001]
    [ .0000000000+00000, .0000000000+00000]

X = [ .5000000000+00001, .5000000000+00001]
    [ .0000000000+00000, .0000000000+00000]
Y = [ .9510347469+00001, .9510347470+00001]
    [ .1518000000-00051, .1518000000-00051]
```

```
X = [ .1000000000+00002, .1000000000+00002]
    [ .0000000000+00000, .0000000000+00000]
Y = [ .2018300000+00002, .2018300000+00002]
    [ .0000000000+00000, .0000000000+00000]

X = [ .1500000000+00002, .1500000000+00002]
    [ .0000000000+00000, .0000000000+00000]
Y = [ .3085654676+00002, .3085654677+00002]
    [ .6490000000-00051, .6490000000-00051]

X = [ .2000000000+00002, .2000000000+00002]
    [ .0000000000+00000, .0000000000+00000]
Y = [ .4108214784+00002, .4108214785+00002]
    [ .3790000000-00051, .3790000000-00051]

X = [ .2500000000+00002, .2500000000+00002]
    [ .0000000000+00000, .0000000000+00000]
Y = [ .5135257125+00002, .5135257126+00002]
    [ .1165000000-00050, .1165000000-00055]
```

MAIRIX_INVERSION

The matrix inversion program finds the inverse of a square matrix. The first example finds the inverse of an Hilbert matrix of order 4. The single precision interval widths were of the magnitude from 10**-3 to 1. The extended interval widths were of the magnitude from 10**-50 to 10**-47.

Single_Precision

```
TIME A TIME AND PAGE FAULTS FOR THE INVERSION ARE AS FOLLOWS:

CPU time =       0.006392 seconds; Page faults =         0

INVERSE OF HILBERT MATRIX OF ORDER 4

ROW     1

   0.16000119e+02   -.12000130e+03   0.24000305e+03   -.14000196e+03

ROW     2
   -.12000130e+03   0.12000141e+04   -.27000332e+04   0.16800212e+04

ROW     3

   0.24000308e+03   -.27000333e+04   0.64800785e+04   -.42000503e+04

ROW     4
   -.14000198e+03   0.16800214e+04   -.42000505e+04   0.28000323e+04
```

Double_Precision

```
TIME AND PAGE FAULTS FOR THE INVERSION ARE AS FOLLOWS:

CPU time =        0.00209 seconds; Page faults =          2

INVERSE OF HILBERT MATRIX OF ORDER 4


ROW      1

   0.160000000000000020d+02          -.120000000000000021d+03
   0.240000000000000049d+03          -.140000000000000031d+03

ROW      2

   -.120000000000000021d+03          0.120000000000000023d+04
   -.270000000000000053d+04          0.168000000000000034d+04

ROW      3
   0.240000000000000051d+03          -.270000000000000054d+04
   0.648000000000000127d+04          -.420000000000000081d+04

ROW      4
   -.140000000000000033d+03          0.168000000000000035d+04
   -.420000000000000082d+04          0.280000000000000052d+04
```

Single Precision Interval

```
TIME AND PAGE FAULTS FOR THE INVERSION ARE AS FOLLOWS:

CPU time =          0.363970 seconds; Page faults =        2

INVERSE OF HILBERT MATRIX OF ORDER 4


ROW 1

   [ .15999083+02, .16000938+02]      [-.12001072+03,-.11998957+03]
         0.9272098541e-03                   0.1057004929e-01

   [ .23997471+03, .24002608+03]      [-.14001772+03,-.13998284+03]
         0.1014995575e-01                   0.1743507385e-01

ROW 2
   [-.12001029+03,-.11998998+03]      [ .11998861+04, .12001176+04]
         0.1014995575e-01                   0.1156997681e+00

   [-.27002862+04,-.26997238+04]      [ .16798125+04, .16801943+04]
         0.2811279297e+00                   0.1980416748e+00
```

```
ROW 3

   [ .23997635+03, .24002433+03]     [-.27002782+04,-.26997313+04]
         0.2398204803e-01                  0.2733917236e+00

   [ .64793482+04, .64806767+04]     [-.42004596+04,-.41995576+04]
         0.6642456055e+00                  0.4509582520e+00

ROW 4

   [-.14001562+03,-.13998484+03]     [ .16798277+04, .16801786+04]
         0.1538562775e-01                  0.1753845215e+00

   [-.42004345+04,-.41995821+04]     [ .27997164+044, .28002950+04]
         0.4261474609e+00                  0.2892761230e+00
```

Extended Precision Interval

```
TIME AND PAGE FAULTS FOR THE INVERSION ARE AS FOLLOWS:


CPU time =        3.018572 seconds; Page faults =      3

INVERSE OF HILBERT MATRIX OF ORDER 4

ROW 1

   [ .1599999999+00002, .1600000001+00002]
   [ .4854000000-00050, .4854000000-00050]

   [-.1200000001+00003,-.1199999999+00003]
   [ .5527000000-00048, .1341800000-00048]

   [-.1400000001+00003,-.1399999999+00003]
   [ .8760000000-00049, .8760000000-00049]

ROW 2

   [-.1200000001+00003,-.1199999999+00003]
   [ .5258000000-00049, .5258000000-00049]

   [ .1199999999+00004, .1200000001+00004]
   [ .5990000000-00048, .5990000000-00048]

   [-.2700000001+00004,-.2699999999+00004]
   [ .1453800000-00047, .1453800000-00047]

   [ .1679999999+00004, .1680000001+00004]
   [ .9493000000-00048, .9430000000-00048]

ROW 3
   [ .2399999999+00003, .2400000001+00003]
```

```
   [ .1249800000-00048, .1249800000-00048]

   [-.2700000001+00004,-.2699999999+00004]
   [ .1423300000-00047, .1423300000-00047]

   [ .6479999999+00004, .6480000001+00004]
   [ .3455500000-00047, .3455500000-00047]

   [-.4200000001+00004,-.4199999999+00004]
   [ .2256400000-00047, .2256400000-00047]

ROW 4
   [-.1400000001+00003,-.1399999999+00003]
   [ .8108900000-00049, .8089000000-00049]

   [ .1679999999+00003,-.1399999999+00003]
   [ .9213000000-00048, .9213000000-00048]

   [-.4200000001+00004,-.4199999999+00004]
   [ .2236900000-00047, .2236900000-00047]

   [ .2299999999+00004, .2800000001+00004]
   [ .1460700000-00047, .1460700000-00047]
```

The next example inverts an Hilbert matrix of order 0. The single precision interval version could not find a solution. To conserve space the extended interval results are not shown but may be found in the attachment to this letter. The extended interval widths ranged from 10**-36 to 10**-28. Also not that the single precision results are meaningless.

Single Precision

```
TIME AND PAGE FAULTS FOR THE INVERSION ARE AS FOLLOWS:

CPU time =      0.065202 seconds; Page faults =        1

INVERSE OF HILBERT MATRIX OF ORDER 10

ROW 1

   0.66628764e+02  -.19763846e+04  0.17260560e+05  -.62370884e+05
   0.95071291e+05  -.29811917e+05  -.52568417e+05  0.12496660e+04
   0.68708063e+05  -.35645043e+05

ROW 2

   -.19894290e+04  0.75388504e+05  -.72811225e+06  0.28178170e+07
   -.46580271e+07  0.20127897e+07  0.23197084e+07  -.10075493e+07
   -.21445800e+07  0.13152878e+07

ROW 3
```

```
   0.17619120e+05  -.73787954e+06  0.76123086e+07  -.31378729e+08
   0.56769210e+08  -.32712767e+08  -.24285263e+08  0.27813182e+08
   0.54030478e+07  -.85081013e+07

ROW 4

   -.65350269e+05  0.29248154e+07  -.32090695e+08  0.14215210e+09
   -.28589484e+09  0.21212882e+09  0.10197568e+09  -.24367823e+09
   0.10984572e+09  -.72728300e+07

ROW 5

   0.10526792e+06  -.50690238e+07  0.60455295e+08  -.29635527e+09
   0.68104005e+09  -.63099936e+09  -.21897011e+09  0.94184239e+09
   -.71255193e+09  0.18048439e+09

ROW 6

   -.42870195e+05  0.25873252e+07  -.38849799e+08  0.23780274e+09
   -.67411450e+09  0.75598362e+09  0.35379525e+09  -.17238461e+10
   0.15614960e+10  -.47486910e+09

ROW 7

   -.58890916e+05  0.25351663e+07  -.25165329e+08  0.94883239e+08
   -.16748904e+09  0.25520953e+09  -.65576277e+09  0.11986492e+10
   -.10311218e+10  0.32840154e+09

ROW 8

   0.35385499e+05  -.24513788e+07  0.41530602e+08  -.28713407e+09
   0.96178677e+09  -.15891991e+10  0.98973335e+09  0.49803286e+09
   -.97000460e+09  0.35773168e+09

ROW 9

   0.36014969e+05  -.71491391e+06  -.91393845e+07  0.16378166e+09
   -.77517486e+09  0.14881434e+10  -.79604687e+09  0.49803286e+09
   0.16805552e+10  -.61501253e+09

ROW 10

   -.25268021e+05  0.85312601e+06  -.36485157e+07  -.26486693e+08
   0.20763149e+09  -.46059757e+09  0.24735363e+09  0.43887222e+09
   -.64172422e+09  0.23783952e+09
```

Double Precision

TIME AND PAGE FAULTS FOR THE INVERSION ARE AS FOLLOWS:

CPU time = 0.104900 seconds; Page faults = 3

INVERSE OF HILBERT MATRIX OF ORDER 10

ROW 1

0.1000000022533443850+03	-.4950000192460866070+04
0.7920000406434825690+05	-.6006000367037112300+06
0.2522520174135427260+07	-.6306300476592509590+07
0.9609600779054336870+07	-.8751600750492890220+07
0.4375800392929251300+07	-.9237800862054949290+06

ROW 2

-.4950000192340670320+04	0.3267000164231968760+06
-.5880600346744003740+07	0.4756752313077461560+08
-.2081079148514068410+09	0.5351346406421998160+09
-.8324316664286631980+09	0.7701408639879927900+09
-.3898838134992774610+09	0.8314020734901691450+08

ROW 3

0.7920000405979467100+05	-.5880600346572172010+07
0.1129075273159667920+09	-.9513504660476412740+09
0.4281077113275985980+10	-.1123782745723314850+11
0.8506555906441023230+10	-.1829084554970765660+10

ROW 4

-.6006000366477685100+06	0.4756752312796014740+08
-.9513504660209916620+09	0.8245037395968065890+10
-.3787564062655199560+11	0.1010017085339165090+12
-.1616027339192626700+12	0.1529079673735490210+12
-.7884317077255198610+11	0.1707145579788619710+11

ROW 5

0.2522520173811493590+07	-.2081079148330695350+09
0.4281077113044379450+10	-.3787564062560242340+11
0.1767529898003569750+12	-.4772330729436236820+12
0.7712857751153455490+12	-.7358695921054678310+12
0.3820861346063978410+12	-.8322334682590271580+11

ROW 6

-.6306300475570232840+07	0.5351346485804550020+09
-.1123782745635542150+11	0.1010017085291161430+12
-.4772330729331796100+12	0.1301544744679337330+13
-.2121035880669498940+13	0.2037792714620523880+13
-.1064382802010233430+13	0.2330253706804890810+12

ROW 7

0.9609600777192597210+07	-.8324316663114912970+09
0.1775854219924699240+11	-.1616027339083218870+12
0.7712857750835950350+12	-.2121035880629325350+13

0.34806742645102340 3d+13	-.33639752722214532 2d+13
0.17660870175121363 9d+13	-.38837561720107403 1d+12

ROW 8

-.87516007485395712 0d+07	0.77014086386150143 6d+09
-.16635042627466375 3d+11	0.15290796736041537 4d+12
-.73586959206257442 2d+12	0.20377927145481384 4d+13
-.33639752721663891 7d+13	0.32678616907994671 6d+13
-.17232864375341711 2d+13	0.38044958369994492 2d+12

ROW 9

0.43758003918330178 0d+07	-.38988381342678519 9d+09
0.85065559052618844 9d+10	-.78843170764481638 3d+11
0.38208613457827492 4d+12	-.10643828019568273 7d+13
0.17660870174579553 0d+13	-.17232864375097479 0d+13
0.91232811319661604 2d+12	-.20211384111426626 6d+12

ROW 10

-.92378008595068480 4d+06	0.83140207331893732 2d+08
-.18290845546863451 3d+10	0.17071455795883297 5d+11
-.83223346818630346 2d+11	0.23302537066575446 7d+12
-.38837561718425355 5d+12	0.38044958368983354 9d+12
-.20211384111177762 0d+12	0.44914186870685418 8d+11

Single_Precision_Interval

SINGLE PRECISION INTERVAL HAS BROKEN DOWN DUE TO DIVISION BY ZERO (see attachment)

Extended_Precision_Interval

The extended interval results are not shown here in order to conserve space. See the attachment for the results.

FFT_Program

The FFT program supplied by WES, once interfaced and running correctly on Multics, was modified to print out the central processor time used and page faults generated during various stages of the overall program. These stages were initialization, the FFT subroutine, and output.

The FFT program was then converted to double precision. The only difficulty encountered during this stage of the work was that complex arithmetic is not supported in double precision in Multics fortran and, therefore, had to be simulated. Similiarly, during the conversion to interval, complex arithmetic again had

to be simulated. Further, in the driver routine, a call to ATAN2 with arguments 1.0 and 0.0 was replaced by the value such a call returns, one half pi. The value returned by ATAN2 was not the minimum interval representation and, when used in subsequent calls to the SIN function, resulted in unacceptably large intervals being returned. The only modification made to the interval version of the program before executing it utilizing the 56 decimal package was to insert a cal to a subroutine, GENPI, to obtain a 56 decimal digit precision value for pi.

The single precision, double precision, interval, and 56 decimal interval versions of the program were executed using the original real coefficients supplied by WES and also using real coefficients in the form of a square wave as suggested by WES. Other real coefficients were also tried but the results added little additional information for analysis.

Using the real coefficients suplied by WES, all complex coefficients but one should have, theoretically, been zero. The following table has example values computed in each of the four different runs.

single precision	0.0265	0.0428	0.2930	0.3623
double precision	0.3113	0.0460	0.3105	0.6678
standard interval	0.8222	2.5133	11.1639	14.4994
56 decimal interval	260.8131	180.6730	25.3920	466.8781

SLOPE Program Work

A large proportion of time had been spent in an attempt to understand the applications programs. The logic was carefully followed using the given data. Throughout the program, the language of the program was updated (FORTRAN IV as opposed to FORTRAN II), detected inefficiencies removed (for example, unnecessary variables, GO TO's to GO TO's, unused labels, GO TO's to the next executable statement, etc.), and, in general, made more readable. This time proved beneficial not only in facilitating the conversion of the routines to interval arithmetic but also in that it exposed some errors in the program (double initialization of some variables, an integer function which should have been a real function, etc.) These errors were corrected and duly reported to WES. It should be noted that all testing was done using the corrected version of the original program rather than the heavily modified version. It was felt that testing conditions should approximate the working conditions at WES as closely as possible.

Once the program was successfully interfaced with Multics and reproducing the desired output, a program of testing was outlined. This consisted of varying a set of three inputs (cohesion, unit weight and phi) for the program plus or minus ten percent, singly and in conjuction with each other. A

comprehensive analysis was run consisting of 81 separate runs with 27 comparisons of 3, and were submitted to WES in July, 1977. The largest problem encountered at this stage was the production of summary reports which presented the analysis in a readable form. The report was finally configured to consist of 27 comparisons of three runs each in regard to central processing unit time, paging and fluctuation of the output data (Table 1).

TABLE 1

Configuration of the 27 comparisons of the test runs. For example, run one compares the three runs cohesion +10% phi +10% unit weight -10%, cohesion +10% phi +10% unit weight -00%, cohesion +10% phi +10% unit weight +10%.

cohesion +10%	phi +10%	unit weight [-10%, no flux, +10%]
cohesion +10%	phi -00%	unit weight [-10%, no flux, +10%]
cohesion +10%	phi -10%	unit weight [-10%, no flux, +10%]
cohesion -00%	phi +10%	unit weight [-10%, no flux, +10%]
cohesion -00%	phi -00%	unit weight [-10%, no flux, +10%]
cohesion -00%	phi -10%	unit weight [-10%, no flux, +10%]
cohesion -10%	phi +10%	unit weight [-10%, no flux, +10%]
cohesion -10%	phi -00%	unit weight [-10% no flux, +10%]
cohesion -10%	phi -10%	unit weight [-10%, no flux, +10%]
phi +10%	unit weight +10%	cohesion [-10%, no flux, +10%]
phi +10%	unit weight -00%	cohesion [-10%, no flux, +10%]
phi -00%	unit weight +10%	cohesion [-10%, no flux, +10%]
phi -00%	unit weight -00%	cohesion [-10%, no flux, +10%]
phi -00%	unit weight -10%	cohesion [-10%, no flux, +10%]
phi -10%	unit weight +10%	cohesion [-10%, no flux, +10%]
phi -10%	unit weight -00%	cohesion [-10%, no flux, +10%]
phi -10%	unit weight -10%	cohesion [-10%, no flux, +10%]
unit weight +10%	cohesion -00%	phi [-10%, no flux, +10%]
unit weight +10%	cohesion -10%	phi [-10%, no flux, +10%]
unit weight -00%	cohesion +10%	phi [-10%, no flux, +10%]
unit weight -00%	cohesion -00%	phi [-10%, no flux, +10%]
unit weight -00%	cohesion -10%	phi [-10%, no flux, +10%]
unit weight -10%	cohesion +10%	phi [-10%, no flux, +10%]
unit weight -10%	cohesion -00%	phi [-10%, no flux, +10%]
unit weight -10%	cohesion -10%	phi [-10%, no flux, +10%]
unit weight +10%	cohesion +10%	phi [-10%, no flux, +10%]

At this time the application programs were converted to double precision and the same testing procedure as outlined above was applied. No significant problems were encountered during the conversion. The output of the two tests were given to the same precision as that given in the report supplied by WES. The summary report along with each run and data was sent to WES for further evaluation.

During the testing it was noted that one of the inputs to be varied, cohesion of the first soil, was zero. As 0-10% = 0+10% =

0 no fluctuation was produced by this parameter. A non-zero value was recieved from WES and the testing procedure applied again. The output was increased to 8 digits of accuracy, the maximum for single precision FORTRAN on Multics, for this set of tests.

An analysis of the test output discerned no significant difference in accuracy between the single and double precision (Table 2).

TABLE 2

Sample values from the case phi-00%, unit weight-00%, cohesion [-10%, no flux, +10%] for single precision (SP), double precision (DP), single precision interval (SPI), and extended interval (EI). The values shown are for CP.

±10%

SP	-1.6056320	-1.3048941	439.65
DP	-1.60563196598609845	-1.30489416517190121	439.65
SPI	[-2.0212812,-1.2585650]	[-1.6226176,-1.0324112]	[435.14,444.21]
EI	[-1.6056320,-1.6056320]	[-1.3048741,-1.3048943]	[439.65,439,65]

no_flux

SP	-1.6069299	-1.3060394	439.63
DP	-1.60692993603933327	-1.30603947523241611	439.63
SPI	[-2.0217068,-1.26044111]	[-1.6231307,-1.0340142]	[435.13,444.18]
EI	[-1.6069299,-1.60692991]	[-1.3060194,-1.3060396]	[439.63,439.63]

=10%

SP	-1.6082275	-1.3071844	439.60
DP	-1.60822753256086878	-1.3071840917743123	439.60
SPI	[-2.0220372,-1.2623842]	[-1.6235718,-1.0356705]	[435.11,444.14]
EI	[-1.6082275,-1.60822751]	[-1.3071643,-1.3071845]	[439.60,439.60]

Better than 50% of the output agreed to the full eight digits, the rest differed by no more than ±3 in the eighth digit. This was attributed to the fact that the Honeywell 68/80 does all floating point computations in double precision.

Concurrent with the above testing was the transformation of the applications programs into interval arithmetic. Several problems occurred during this period which greatly hampered progress. The first problem was that, apparently, the given interval routines to read in data were designed to read from one file without interruption, i.e. file 5 in FORTRAN. The applications and then reading from it. The second problem, which interacted with the first to create a much larger problem, was that a bug in the

Multics FORTRAN I/O was struck upon. Normally when a record of, say, 256 characters is read from a file containing 80 characters, the remaining 176 characters are padded with blanks. In this case they were not; an error message, "record too short" was produced instead. These problems were solved and work begun on the interval version of the testing.

Testing procedures for the interval version were the same as those for single and double precision. During the testing of the interval version two problems worth noting were encountered. The first problem concerned the manner in which the algorithm was coded. The second problem involved the identification of data sensitivity. This was much more difficult to handle, taking some effort even to determine the nature of the source. Once the nature of the problem was discovered it took an even larger portion of time to track down the source of the problem owing to the near impossibility of following the logic of the programs. The first problem was a result of the way in which the interval package evaluates the test value in an arthmetic IF statement. When an arithmetic IF is encountered with an interval test value the interval is converted to a real (i.e. the midpoint is taken) and the branch evaluated as normal. The difficulty was to be taken only if the test value was positive. Certain intervals were passing along this branch whose midpoint was inneed positive but whose left endpoint was negative. The interval was subsequently used as a divisor; as it contained zero a zero-divide error was flagged by the interval arithmetic routines. The problem was solved simply by recoding the branch as a logical IF statement which is evaluated in a different manner and avoids this problem.

The second problem, as stated before, was much more difficult to handle. During the testing it was noted that some of the runs contained intervals which were "blowing up", that is, the width of the intervals were becoming quite large. After a perusal of the output a correlation was discovered between the blownup intervals and the varying of unit weight by -10%: all runs which varied unit weight by -10% contained blownup interals! Desiring to know with some certainty at what point the intervals would start blowing up the following strategy was devised: each run would start out with unit weight varied by zero; unit weight would then be decremented by units of .25% of the initial value until the intervals would blowup (Table 3). Once this strategy was carried out a value (2.25%) was found at which unit weight could be decreased without generating large interval widths. The testing procedure was redone using this value to decrease unit weight. In the summary report generated for this set of testing a note is made indicating those runs in which the unit weight is decreased by this value rather than the normal 10%.

TABLE 3

Sample values from one of the graduated runs (cohesion-00%, phi-10%). T3 = FS1-FSL. For this run the intervals became unstable at unit weight-3.25%. (* indicates infinity)

UNIT WEIGHT - 2.50%
T3 = [-.382388e-01,-.380500e-01]
CP = [-1.786,-1.094] [-1.43,-897] [435,444]

UNIT WEIGHT - 2.75%
T3 = [-.38430e-01,-382241e-01]
CP = [-1.786,-1.093] [-1.43,-895] [435.444]

UNIT WEIGHT - 3.00% (bounds faults occurred)
T3 = [-.139087e-03,0.826716e-04]
CP = [-1.977,-.9748] [-1.57,-.801] [434,446]

UNIT WEIGHT -3.25% (bounds faults occurred)
T3 = [-.240356e-03,-.183582]
CP = [-*,+*] [-*,+*] [34,5731]

While these runs were being made a large amount of time and effort was spent in the tracking down of the source of the expanding intervals. After an extended effort the source was traced with a large degree of confidence to one statement, "T3 = FS1 - FSL". It seems that as unit weight is decreased the difference between FS1 and FSL becomes increasingly small. After a time the subtraction has the effect of stripping off the significant digits of accuracy of the resultant interval. The problem was compounded by using T3 as a divisor in a subsequent computation thus exploding the interval during the next few computations.

One other unexpected benefit was reaped by during this procedure. While tracing through the routines it was noted that in the subroutine WGHT several computations could be combined and an interval consistently of less than optimal width could be factored out producing a more accurate algorithm. While its disturbing influence could not altogether be avoided it was minimized.

The 56 decimal digit version of the interval package was then made available. Testing was done as before. Three outstanding features of the testing were noted. The first was that the data dependency of the algorithm as noted above disappeared, the interal widths getting no larger than 10**-4. The second was the overly large amount of central processor unit time required for processing each run (55 minutes plus or minus 6 minutes) (Table 4). It was noted, however that most of this time was spent in just a few of the interval routines, principally the trigonometric functions which take considerable amounts of time to evaluate when the argument is greater than one. The fast fourier transform routines did not encounter this large an

increase in the amount of central processing unit time per run.

TABLE 4

This table shows the maximum and minimum central processor unit times which were encountered during each set of testing runs for single precision, double precision, single precision interval, extended interval.

SINGLE PRECISION
max -- 2.80 (cohesion+10%, phi-10%, unit weight-10%)
min -- 2.48 (cohesion-00%, phi-10%, unit weight-00%)

DOUBLE PRECISION
max -- 3.08 (cohesion+10%, phi-10%, unit weight-00%)
min -- 2.58 (cohesion-00%, phi+10%, unit weight-00%)

SINGLE PRECISION INTERVAL
max -- 26.30 (cohesion+10%, phi-10%, unit weight+10%)
min -- 22.35 (cohesion-10%, phi-10%, unit weight-10%)

EXTENDED INTERVAL
max -- 3719 (cohesion-00%, phi-00%, unit weight+10%)
min -- 2951 (cohesion-00%, phi-10%, unit weight-00%)

The third and most pleasant of the notable features was the complete lack of problems in the bringing up and testing of the 56 decimal digit version of the algorithm. This was a benefit of the absence of any needed large modifications to be made to the single precision interval to convert it to the 5 decimal digit version. The only required modification was that of a slight adjustment to the output parameters to widen the output field. This was required as the exponent field supplied by the 56 decimal digit routines was somewhat larger.

STRESS

The STRESS program finds the stress on a plane at particular nodal points. This program was provided to us as an extra program from WES to analyze. The program was run in single precision and double precision. The program terminated abnormally in both cases. The output produced by WES also indicated that the program terminated abnormally. The output produced at USL matched the output provided by WES up to the point of termination. The program was also run in single precision interval and extended interval and also terminated abnormally. We are currently waiting to recieve from WES corrections to the STRESS program and additional input to run the program again.

Conclusions and Recommendations

We have concluded that the use of single precision and 56 decimal digit extended precision interval arithmetic can, at times, be extremely useful. It can be used to show the limits of precision of an algorithm. From the testing it was shown that when using the 56 decimal digit data type much better bounds were obtained for the results than when using the single precision interval data type. This was expected for two reasons: 1) 56 decimal digits carry more precision than 27 binary digits (approximately equivalent to 8 decimal digits) used for single precision and 2) there was no conversion error on input and output. The price paid for this increase in precision is a decrease in runtime efficiency. The testing indicated that single precision interval arithmetic resulted in, at most, one order of magnitude increase precision execution. 56 decimal digit interval operation resulted in a further increase of more than one to more than two orders of magnitude.

One application for 56 decimal interval arithmetic would be to validate existing routines. Any data sensitivity discovered could be included in a description of the algorithm and directions on its use. Although 56 decimal interval arithmetic is expensive, its cost must be balanced against possible consequences of using invalid results. A defective dam or the moving of 100,000 tons of dirt unnecessarily would cost considerably more than a few hours of computer time.

A more cost effective technique might be to first test the algorithm using single precision interval arithmetic. Its relatively small decrease in runtime efficiency indicates that its use is more than justified as an economical means in identifying possible trouble areas in an algorithm for the data under consideration. The more expensive 56 decimal interval arithmetic could then be used to investigate those cases with possible problem areas.

Interval arithmetic can also be used to determine the precision of the arithmetic required to guarantee a given precision in the results of an algorithm. In some of the benchmarks executed in 56 decimal digit interval arithmetic, the results were good only to 40 or so digits. This represents a considerable loss of precision. It also points out why arbitrarily picking a given precision for arithmetic does not guarantee results in which absolute confidence can be placed. How great an increase in precision is obtained, if any, by going from a machine with 32 bit words to one with 60 bit words?

In general, whether using interval or regular arithmetic, the greater the precision the longer the run time required for a given algorithm. Having variable precision interval arithmetic would allow the validation of algorithms for which single precision interval arithmetic is insufficient without having to go all the way to 56 decimal digit precision. There will also be instances where it might be desirable or necessary to go beyond

56 decimal digits of precision. There will also be instances where it might be desirable or necessary to go beyond 56 decimal digits of precision. In any case, the overhead associated with execution in interval arithmetic will only be as great as required for the necessary precision.

The execution speed of interval arithmetic can be increased in several ways. One would be to decrease the number of levels of interpretation required in the current implementation. The optimum solution would be to have a hardware or firmware module which could execute variable precision interval arithmetic. (Many existing minicomputer systems have undefined opcodes and ports for just such requirements). As a side effect, an arithmetic unit that can execute variable precision interval arithmetic can also execute variable precision regular arithmetic. This means that interval arithmetic, of the necessary precision, could be used to determine the required arithmetic precision required for the results of the algorithm. Then, the algorithm could be executed using only the required precision on the special arithmetic module.

References

[1] Brent, R. P., "A fortran multiple-precision arithmetic package," Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, May, 1976.

[2] Ladner, T. D. and Yohe, J. M., "An interval arithmetic package for the Univac 1108," The University of Wisconsin, Mathematics Research Center, Technical Summary Report No. 1055, may, 1970.

[3] Reuter, E. K., Podlaska-Lando, S. and Shriver, B.D., "The implementation of the mathematics research center's interval arithmetic package on the MULTICS system," Computer Science Department Report 76-7-2.A, University of Southwestern Louisiana, Lafayette, Louisiana August, 1976.

[4] Yohe, J. M., "Best possible floating point arithmetic," The University of Wisconsin, mathematics Research Center, Technical Summary Report No. 1054, March, 1970.

Report No. 76-7-2.A*
August, 1976

The Implementation of the Mathematics Research Center's Interval Arithmetic Package on the MULTICS System**

Computer Science Department
University of Southwestern Louisiana
Lafayette, Louisiana 70504

by

Eric Reuter, S.Podlaska-Lando, B.D.Shriver

* This report superceeds Report No. 76-7-2

** This work was supported under contract DACA39-76-M-0249 from Department of the Army, Computer Analysis Branch, Waterways Experiment Station, Corps of Engineers.

Table of Contents Page

1.0 General considerations on the computer implementation of interval arithmetic

The floating number system used on computers is an approximation to the real number system. In interval arithmetic, real numbers are approximated by intervals which contain the number. A brief introduction to interval arithmetic is given in Appendix A. We will represent an interval as a pair of floating point numbers stored in consecutive storage locations. The first number will be the left endpoint and the second number will be the right endpoint of the interval. Since the floating point system used on computers is an approximation to the real number system, there are many intervals whose endpoints do not have an exact representation in a particular floating point system. In this case the endpoints of the interval have to be approximated by the floating point system.

We will regard intervals as bounds on an exact but unknown real number. We would like the computer approximation to the interval to also bound the same real number. This means that if the interval $[a',b']$ is a computer approximation to the interval $[a,b]$, then we would like $[a,b] \subseteq [a',b']$. In order to insure that the preceeding set inclusion always holds, a' must be a lower bound for a and b' must be an upper bound for b. Since we want the best computer approximation to the interval, we want a' to be the greatest lower bound for a and b' to be the least upper bound for b. In this way the interval $[a',b']$ will be the smallest computer representable interval that contains $[a,b]$.

In order to obtain the smallest computer representable interval for the result of arithmetic operations on intervals, directed roundings on the computer artihmetic operations must be defined. If x is a real number and $M1$ and $M2$ are two consecutive machine representable numbers such that $M1<x<M2$ and if r is a rounding function, then r is downward directed if $r(x) = M1$ and r is upward directed if $r(x) = M2$. $M1$ and $M2$ will be the machine representable numbers that are respectively the greatest lower bound and the least upper bound for the real number x. If x is a machine representable number, then $r(x) = x$.

In general a op b, where a and b are machine representable numbers and op is one of the machine arithmetic operations, is not a machine representable number and must be rounded into a machine representable number. Algorithms for performing the machine arithmetic operations with directed roundings can be found in Yohe [2]. These operations are used to compute the endpoints of the resultant interval for a particular arithmetic operation performed on two intervals. A downward directed rounding is performed on the left endpoint and an upward directed rounding is performed on the right endpoint.

For example, in Appendix A, interval addition is defined as follows:

$$[a,b] + [c,d] = [a+c,b+d]$$

We assume now that a, b, c, and d are machine representable numbers. The computer approximation to the resultant interval is defined as follows:

$$[a,b] \oplus [c,d] = [r1(a \oplus c), r2(b \oplus d)]$$

where $\oplus$ is the machine addition operation and r1 is a downward directed rounding and r2 is an upward directed rounding.

Since the exponent range is bounded, certain faults may occur during an arithmetic operation. If the exponent becomes too small, underflow has occurred. If the exponent becomes too large, then overflow has occurred. If underflow occurs, then the true result is between zero and the smallest positive or negative representable number. In this case a directed rounding can give a valid bound. In the case of overflow, if rounding away from zero is wanted, then there is no machine representable number which can be used as a correct bound. This type of fault is known as an infinity fault.

1.1 Interval Valued Functions

A real-valued function, f, which is defined and continuous on an interval [a,b] can be extended to an interval-valued function, F, of an interval variable by defining

$$F([a,b]) = \{f(x) : x \in [a,b]\}.$$

When f is evaluated on a digital computer using machine representable approximations to the real numbers, a computer approximation, f', to f results. If F([a,b]) is an interval valued function of an interval (where a and b are machine representable numbers), then the computer approximation, F'([a,b]) is defined as an interval that contains F([a,b]).

If f' is the computer approximation of a real valued function f and f is monotonic increasing on [a,b], then

$$F'([a,b]) = [r1(f'(a)), r2(f'(b))]$$

where r1 is a downward directed rounding into a machine representable number such that $r1(f'(a)) \leq f(a)$ and r2 is an upward directed rounding into a machine representable number such that $r2(f'(b)) \geq f(b)$. Ideally we would like r1(f'(a)) to be the largest machine representable number such that $r1(f'(a)) \leq f(a)$

(i.e., a greatest lower bound) and r2(f'(b)) to be the smallest machine representable number such that r2(f'(b)) ≥ f(b) (i.e. a least upper bound).

If f is monotonic decreasing on [a,b], then

$$F'([a,b]) = [r1(f'(b)), r2(f'(a))]$$

If f is not monotonic on [a,b], then the interval [a,b] can be divided into disjoint subintervals; X'(i), i = 1,2,3,...,n; where the endpoints of each X'(i) are machine representable numbers and U X(i) contains all the machine representable numbers in the interval [a,b] and f is monotonic on each X'(i). In this case F'([a,b]) = U F (X'(i)).

It may not be possible to obtain the best bounds for the result of the computer approximation to the function f. The problem will be illustrated in the next section when describing the interval counterparts of the MULTICS basic external functions.

2.0 The Implementation of the MRC Interval Arithmetic Package for the MULTICS System

The interval arithmetic package and the input/output routines for interval numbers which are implemented on the MULTICS system follow the design of an interval arithmetic package implemented on the UNIVAC 1108 computer located at the Mathematics Research Center of the University of Wisconsin [1,3]. This section mainly presents the difficulties encountered when the interval arithmetic package was implemented on the MULTICS system and also the changes that were made to the original interval package implemented at MRC. Most of the changes dealt with machine dependencies.

Before the routines are described, a description of the representation of interval numbers on MULTICS will be given along with a description of the MULTICS double precision floating point format and how it impacted the realization of the interval package. The endpoints of the intervals are represented as a pair of floating point numbers stored in consecutive storage locations. The MULTICS single precision floating point format uses a 36 bit word which consists of an 8 bit 2's complement exponent, with the high order bit the sign bit, followed by a 28 bit normalized 2's complement fraction, with the high order bit the sign bit.

In the original interval package implemented on the UNIVAC 1108, the type double precision in Fortran was used extensively to trap underflow and overflow fault conditions. This could be done because the exponent range of the double precision floating point

format on the UNIVAC 1108 is greater than the single precision floating point format. Therefore, with certain precautions, results could be computed in double precision without fear of machine underflow or overflow. The underflow or overflow could then be trapped when the conversion is made to single precision. The MULTICS double precision floating point format uses a 72 bit double word which consists of an 8 bit 2's complement exponent, with the high order bit the sign bit, followed by a 64 bit normalized 2's complement fraction, with the high order bit the sign bit. The double precision floating point format has the same exponent range as the single precision format. Therefore, it is much more difficult to trap certain faults. The problem is amply illustrated in the section describing the implementation of the interval basic external functions.

2.1 Interval Arithmetic Package Subroutines

The subroutines of the MRC interval package can be divided into eight categories. These categories are arithmetic operations, exponentiation operations, conversion functions, comparison, basic external functions, supporting functions, input/output routines and miscellaneous. All of the routines in each category except the input/output category were written in Fortran at the upper level. Several of the Fortran subroutines call routines that are written in PL/I. These PL/I routines correspond for the most part to the assembler routines that were written for the UNIVAC 1108 version of the interval package and are written specifically for the MULTICS implementation. Most of the input/output routines were written in PL/I.

Each subroutine in the package will be described briefly except when changes had to be made to a particular subroutine because of some machine dependencies. In that case a more detailed description will be given. After the description of each routine a code will be given that specifies whether that particular routine was implemented in its original form from the UNIVAC 1108 version, or it was implemented with some changes from the UNIVAC 1108 version, or it was written in PL/I specifically for the MULTICS system. The codes are [MRC] for the original form with no changes, [MRC/M] for the original form modified, and [MUL] for the routines written specifically for the MULTICS system. Appendix C provides a summary of the routines in each of the above categories. A complete source listing of the MULTICS interval package can be found in [8].

2.1.1 Arithmetic Operations

The routines in this category perform the four basic arithmetic operations of addition, subtraction, multiplication, and division on interval numbers. Since we want the best computer approximation to the results of computer arithmetic operations on intervals, directed roundings on the computer arithmetic operations must be performed. The floating point hardware on the MULTICS system does not perform directed roundings. Therefore the four basic single precision floating point computer arithmetic operations of addition, subtraction, multiplication, and division had to be simulated in order to provide the correct roundings. A description of the routines that simulated the floating point computer arithmetic operations and provided the proper directed roundings and a description of the routines that perform the basic computer arithmetic operations on intervals follows. These routines perform the "best possible arithmetic" computer operations with directed roundings as described by Yohe [2]. All the routines are written in PL/I for the MULTICS system.

bpaadd: Performs single precision floating point addition. [MUL]

bpasub: Performs single precision floating point subtraction. A problem can occur in this routine because the subtraction is realized by negating the second operand and calling the bpaadd routine. The problem is negating the smallest positive representable number, because underflow will occur using the MULTICS floating point hardware. This problem can occur in general and the solution to the problem is described below in the description of the pack routine. [MUL]

bpamul: Performs single precision floating point multiplication. [MUL]

bpadiv: Performs single precision floating point division. [MUL]

brounding: Performs bounds checking and rounding on the results from the above four arithmetic operations. The rounding strategies employed are toward zero, away from zero, downward directed, upward directed, and optimal. [MUL]

unpack: This routine unpacks the floating point number from MULTICS format into a format that the bpa routines will handle. The number is made positive because the bpa routines perform their operations on signed magnitude fractions. [MUL]

normalize: This routine is used by the bpa routines to normalize the fraction. [MUL]

shift_right: This routine is used by the bpa routines to shift the fraction right when fraction overflow occurs. [MUL]

s_mgn_add: This routine performs a signed magnitude addition of two 36 bit binary integers. It is used by the bpaadd routine to add the two fractions. [MUL]

pack: This routine packs the floating point result produced by the bpa routines into the MULTICS floating point format. A problem can occur in this routine when the number to be packed is the negative of the smallest positive representable number. The bpa routines perform signed magnitude arithmetic on the fractions and a negative result is obtained in MULTICS format by negating the positive result. The negation of the smallest positive representable number will cause an underflow using the MULTICS floating point hardware because the normalized form of that number has an exponent of -129 and is therefore not representable. If the number is to be negated and it is the smallest positive representable number, then the bit pattern that represents the negative of the smallest positive representable number is assigned to the result. This number is not in true 2's complement normalized form, but represents the true value. [MUL]

The following two Fortran subroutines perform the arithmetic operations of addition, subtraction, multiplication, and division on intervals. The routines call the bpa routines described above to perform their operations on the endpoints of the intervals.

arith1: This routine performs the operations of addition and subtraction on intervals. There is an entry point for addition and an entry point for subtraction. The operations are performed on the endpoints as described in Sections 1 and 2. A slight change was made in this routine from the original routine implemented in the UNIVAC 1108 version of the interval package. The original routine only made calls to the bpaadd routine. In the case of interval subtraction the endpoints of the second interval operand were negated and the bpaadd routine was then called. This negation could cause the same problem as described above in the bpasub and pack routines. Since the problem had been taken care of in the bpasub routine, it was decided to call the bpasub routine directly if an interval subtraction was to be performed. [MRC/M]

arith2: This routine performs the operations of multiplication and division on intervals. There is an entry point for multiplication and an entry point for division. As is stated in Appendix A, the signs of the endpoints of the intervals being multiplied or divided are examined in order to determine in advance which products or quotients will be the maximum and minimum. [MRC]

2.1.2 Exponentiation Operations

The routines in this category perform various exponentiation operations involving interval, double precision, real, and integer numbers. The exact nature of the exponentiation performed will be described in the description of each Fortran routine that follows:

expon1: This routine calculates the value of an interval raised to an integer power. [MRC]

bpaxp4: This routine computes the best value of a real number raised to an integer power. It is used by expon1 to calculate the value of an interval raised to an integer power. Changes were made in this routine to correct the situation in which the fault flag may not be set correctly and to take care of the problem of negating the smallest positive representable number. [MRC/M]

expon2: This routine calculates base ** power where base is an interval number and power is real, double precision, or interval. There is an entry point for each type of exponentiation. [MRC]

2.1.3 Conversion Functions

The routines in this category perform conversions from the standard types to type interval and from interval to the standard types. The following routines are written in Fortran.

convrt: This routine has entry points to convert from integer to interval, complex to interval, real to interval, and double precision to interval. [MRC]

intc84: This routine converts from interval to integer. A change was made in the routine to set the sign of the result correctly when the maximum result needs to be set. [MRC/M]

intc85: This routine converts from interval to real. Changes were made in this routine to check for underflow

differently than in the original due to the fact that the exponent range of a double precision floating point number is not greater than the exponent range of a single precision floating point number on the H68/80 computer. [MRC/M]

intc86: This routine converts from interval to double precision. Changes were made in this routine to check for underflow differently than the original because of the same exponent range of the double precision and single precision floating point formats. [MRC/M]

intc87: This routine converts from interval to complex. [MRC]

funct3: This routine computes an interval with integer endpoints (in floating point form) which contains the interval of the argument. [MRC]

2.1.4 Comparisons

The routines in this category are the relational intrinsic functions for type interval. The following routine was written in Fortran.

relatn: This routine has entry points for the relational functions of equal, not equal, less than, less than or equal, greater than, and greater than or equal. [MRC]

2.1.5 Basic External Functions

Included in the interval package are the interval counterparts of the MULTICS basic external functions atan2, exp, alog, alog10, sin, cos, tan, asin, acos, atan, sinh, cosh and sqrt. The general method of calculation of the interval functions involves bounding the results of the corresponding double precision basic external function. For functions that are monotonic over an interval, the endpoints of the resultant interval are the result of the double precision function evaluated at the endpoints of the input interval and then properly bounded. If the function is not monotonic over the interval, then a case analysis is done by dividing the input interval into subintervals over which the function is monotonic.

The result obtained from the double precision functions must be bounded before it can be used as the endpoint of an interval. Therefore, the accuracy of the results of the double precision basic external functions are required by determining a lower bound on the number of bits of the fraction that the result is

guaranteed to have. the number of bits of the fraction that the result is guaranteed to have are required. This can be illustrated by the following example. Suppose a result is accurate to 35 bits of fraction and a 27 bit lower bound for the result is required. Assume that the 27th through 37th bits of the fraction were 10000000000. If the result were just truncated to 27 bits the 27th bit would be a 1. If however the 37th bit was one unit too large, then bits 27 through 37 would be 01111111111 and the 27th bit of the correct lower bound would be 0. It cannot be determined which case is correct.

The following general bounding technique is performed which will produce correct bounds in all cases, but not necessarily optimal bounds. If a lower bound is sought for the double precision result, then the fraction is decremented by one at or before the last bit known to be accurate. If an upper bound is sought, then the fraction is incremented by one at or before the last bit known to be accurate. The same bounding technique used in bounding the results of the arithmetic operations is then used to obtain the 27 bit fraction of the result.

The following Fortran routines compute the basic external functions for the interval type.

funct2: This routine has entry points for the following interval functions: sqrt, log, exp, log10, atan, asin, acos, tanh, sinh, cosh. Changes made to the asin and acos functions are described in Section 2.1.5.2 and changes made to the functions sinh, cosh and tanh are described in Section 2.1.5.3. [MRC/M]

funct4: This routine calculates sin(arg) and cos(arg) where arg is an interval. There is an entry point for the sine and an entry point for the cosine function. The cosine routine scales the argument so that the left endpoint is in the interval [0,2pi], and then performs a case analysis. The sine routine performs the same scaling as the cosine routine and performs a case analysis. [MRC/M]

funct5: This routine calculates tan and atan2 of an interval. The entry point is provided for either function. The tangent is calculated as follows: the argument is reduced so that the left endpoint is in the interval: [-pi/2, pi/2] and then a case analysis is performed. The atan2 routine takes two interval arguments, x and y and computes atan(x/y). [MRC/M]

2.1.5.1 Accuracy testing

To our knowledge, there is no documentation concerning the implementation of the basic external functions on MULTICS

accessible either by PL/I or Fortran. We considered three approaches to determining the accuracy of the required external function:

1) rigorous error analysis of current implementation
2) rewriting of the required routines
3) comparison of accuracy with known test data

First, the error analysis of the mathematical library routines seemed to be impossible due to the a) lack of description of the algorithms employed, b) low readability of the source programs (much of which was written in ALM - Assembly Language of MULTICS). The second possibility had to be eliminated due to the time constraints of the project and therefore the third approach had to be taken.

The testing itself was done in two stages:

stage1 - generation of input test data and evaluation of the given function

stage2 - comparison of significant digits of the result and corresponding value in the tables

"Driver" programs were written which generated test data and called the routines which were to be tested. The standard tables of functions, i.e. Handbook of Mathematical Functions by Milton Abramowitz and Irene A.Stegun [7] were used. The output was generated in decimal form and then a check was made as to the first digit that was different from the result given in the table. All digits of function values which were tested proved to be identical with corresponding digits in the Handbook. The only exception being the last digit in the Abramovitz's tables. However, the analysis of the very next digit in our results showed that in each case the error was caused by an upward rounding.

The test data had been restricted to the decimal values that can be represented exactly in the floating binary notation. Thus, we avoided the input conversion error and the function value could be obtained for the true argument. Also, we have to warn that the accuracy estimated in this way must be somewhat pessimistic. We were able to check only as many digits as were given in the standard tables. Thus, the tan function is assumed to have only 8 accurate decimal digits even though there are reasons to believe that accuracy is much greater than that.

The list of the number of decimal digits (and binary estimates as well) that are assumed to be accurate is given below.

	Accuracy	
Function	decimal	binary
sqrt	10	33
log	16	52
log10	10	33
exp	16	52
sin	17	56
cos	17	56
tan	8	27
asin	12	39
acos	12	39
atan	12	39
sinh	9	29
cosh	10	33
tanh	8	27

2.1.5.2 Conditions for errors

The Univac 1108 double precision word has an 11 bit exponent field vs. an 8 bit exponent in the single precision word. This allowed the checking for overflow and underflow faults to be done during the conversion from double to single precision format, as was stated in section 2.0. This strategy was not applicable for MULTICS due to the same size of the exponent in both the double and single precision format. In conclusion, the check for eventual fault conditions had to be made prior to the calls to the double precision functions.

The following functions could produce overflows: exp, sinh, cosh and tanh. In the MULTICS implementation, the overflow was prohibited by restricting the allowable domain of the argument to the interval [-88.028,88.028]. From this it followed that for arguments x such that abs(x)>88.028, a special action had to be taken. At this point it turned out, that the magnitudes of results produced at the endpoints were much smaller than the largest representable number. This implied that the actual domain should be extended beyond [-88.028,88.028]. The attempt was made to compute the new endpoints and either compute (if possible) or estimate the proper bounds for the left and right endpoints of the interval result. The detailed discussion of these cases will be given later on.

The occurence of underflow was detected in the double precision functions asin and acos. Analysis of the source programs revealed that an underflow condition was raised at the point of the internal function call to the atan routine. Namely,

```
asin(x) = atan(x,sqrt(-x*x+1))
acos(x) = atan(sqrt(-x*x+1),x)
```

and for x very small, the multiplication operation caused an underflow. The overflow and underflow fault conditions have been tested with a number of programs.

2.1.5.3 Domain extension for functions exp, sinh, cosh.

As we mentioned before, the MULTICS implementation of the functions exp, sinh and cosh restricts the domain to the interval [-88.028,88.028] . Let MIN denote the smallest positive , and MAX the largest positive machine representable number. The endpoints that could actually cause overflow or underflow were obtained from:

$$\log(MIN) = -89.415$$
$$\log(MAX) = 88.029$$

Thus, the domain of exp could be extended to the interval [-89.415,88.029] with exp evaluating to MIN or MAX at the endpoints. The value of exp outside of this interval and in the intervals [-88.415,-88.028] and [88.028,88.029] was evaluated as follows:

lep or rep	explep	exprep
x < -89.415	MIN, underflow	MIN
x = -89.415	MIN	MIN,round up
-89.415 < x < -88.028	MIN	exp(-88.028)
-88.028 <=x<= 88.028	exp(x),round down	exp(x),round up
88.028 < x < 88.029	exp(88.028)	MAX
x = 88.029	MAX,round down	MAX
x > 88.029	MAX, overflow	MAX, infinity

where "round down" means round to the next smaller machine representable number and "round up" means round to the next larger machine representable number.

The largest values of the functions hyperbolic sine, cosine, and tangent could be computed for the argument x=88.029 (since exp(88.029)=MAX). However, even then they were much smaller than the largest representable number. Let x denote the left or the right endpoints of the argument. For x very large we have

$$\sinh(x) = \exp(x)/2$$
$$\sinh(-x) = -\exp(x)/2$$
$$\cosh(x) = \exp(x)/2$$

The smallest positive argument that would cause an overflow was obtained from:

$$\exp(x)/2 = MAX \Rightarrow \exp(x)/2 = \exp(88.029) \Rightarrow x = 88.029+\log(2)$$

The optimal bounds for the left and right endpoint are shown in the table below. "lep" and "rep" denote the left and right endpoints of the interval argument.

hyperbolic_sine

lep or rep = x	sinhlep	sinhrep
x > 88.029+ln(2)	MAX, overflow	MAX, infinity
88.029 < x<=88.029+ln(2)	MAX/2	MAX
x = 88.029	MAX/2, round down	MAX/2
88.028 < x <88.029	sinh(88.028)	MAX/2
-88.029< x <-88.028	-MAX/2	sinh(-88.028)
x = -88.029	-MAX/2	-MAX/2, round up
-88.029-ln(2)<=x<-88.029	-MAX	-MAX/2
x < -88.029-ln(2)	-MAX, overflow	-MAX, infinity

hyperbolic_cosine

lep or rep = x	coshlep	coshrep
x >= 88.029+ln(2)	MAX, overflow	MAX, infinity
88.029<x < 88.029+ln(2)	MAX/2	MAX
x = 88.029	MAX/2	MAX/2,round up
88.028 < x < 88.029	cosh(88.028)	MAX/2

hyperbolic_tangent

lep or rep = x	tanhlep	tanhrep
x < -88.028	-1.0	-1.0,round up
x > 88.028	1.0,round down	1.0

2.1.6 Supporting Functions

The following routines perform some useful functions involving intervals. All the routines are written in Fortran.

funct1: This routine has entry points to calculate the absolute value of an interval, to store the value of one interval into another and to store the negative of one interval into another. A change was made to this routine to take care of the problem of negating the smallest positive representable number. [MRC/M]

supinf: This routine has an entry point that returns the left endpoint of an interval and an entry point that returns

the right endpoint of an interval. [MRC]

unints: This routine has an entry point that returns the union of two intervals and an entry point that returns the intersection of two intervals. [MRC]

length: This routine returns the length of an interval. A change was made in this routine to compute the length differently than in the original. The length is computed by performing a single precision subtract with an away from zero rounding strategy. This change was made in order to trap underflow or overflow. [MRC/M]

intbnd: This routine returns an interval which bounds a double precision value to a specified accuracy. [MRC]

dist: This routine computes the distance between two intervals. [MUL]

compos: This routine returns an interval that consists of the two real arguments as endpoints. [MUL]

2.1.2 Input/Output Routines

The routines in this section were designed to some extent after the I/O routines implemented for the UNIVAC 1108 version of the interval package [3]. Additional routines were included in the MULTICS version to handle scalar interval variables and a matrix of interval variables. A brief description of each I/O routine is given here. In Section 3, which describes writing interval Fortran programs, a more detailed description of the routines is provided giving the calling sequence and several examples for each routine. All of the following routines are written in PL/I.

intrdv: This routine reads interval numbers into any number of interval scalar variables. [MUL]

intrdf: This routine reads interval numbers into an interval vector. [MUL]

intrdm: This routine reads interval numbers into an interval matrix. [MUL]

intprv: This routine outputs interval numbers from any number of interval scalar variables. [MUL]

intpr: This routine outputs interval numbers from an interval vector. [MUL]

intprm: This routine outputs interval numbers from an interval matrix. [MUL]

The following routines are the supporting routines needed by the above routines in order to perform interval I/O. All the routines are written in PL/I except where noted.

convert_to_binary: This routine converts from fixed decimal to floating binary performing a specified rounding. [MUL]

convert_to_decimal: This routine converts from fixed binary to floating decimal. [MUL]

convert_fb_dec: This routine converts from floating binary to floating decimal performing a specified rounding. [MUL]

get_next_int_number: This routine reads the next interval number in the input stream making a syntax check of the number. [MUL]

round_dec: This routine performs a specified rounding of a decimal number. [MUL]

get_char: This routine returns the next character in the input stream. This routine is used by the get_next_int_number routine. It is written in Fortran. [MUL]

set_input_pointer: This routine sets the input pointer for the get_char routine in order to start it off. [MUL]

2.1.8 Miscellaneous

The following routines either fit no other category and/or are used by routines in more than one category.

intrap: This PL/I routine traps all faults that can occur during an operation involving interval operands. It is called by practically all the routines in the package after an operation is performed involving interval numbers. The intrap routine displays an appropriate error message and any arguments and then takes some action depending on the type of fault that occurred. The action taken by intrap when a fault occurs can be specified by the user or a set of default actions can be taken. Appendix B lists all the faults that can occur and the default action taken by intrap after a fault has occurred. Appendix B also

provides a description of how the user can change the action that intrap takes after a particular fault has occurred. [MUL]

bpac68: This PL/I routine converts a double precision number to a single precision number using a specified rounding strategy. The double precision number is taken to be exact. [MUL]

bpac98: This PL/I routine converts a double precision number to a single precision number where the accuracy in number of bits of the fraction of the double precision number is given. [MUL]

intas: This PL/I routine is used to allow the assignment of interval constants to interval variables in a Fortran program. A description of how this is done can be found in Section 3 describing the writing of interval Fortran programs. [MUL]

set_common: This Fortran routine is used to set up the default actions taken by the intrap routine after a fault has occurred. [MUL]

finish: This Fortran routine closes the standard Fortran input and output files and stops the program. It is called by intrap when the action specified for a particular fault is to halt the program. [MUL]

comput: This Fortran routine is used to compute the interval result of an interval function. It is called by the interval basic external function routines. [MRC]

aidint: This PL/I routine is used to return the double precision integer portion of its double precision argument. [MUL]

3.0 Writing Interval Fortran Programs on MULTICS using Interval Arithmetic

This section provides the user of the interval package with a guide to writing interval Fortran programs. An interval Fortran program is a Fortran program in which the extended data type interval is used. After an interval Fortran program has been written, it must be processed by the AUGMENT precompiler [4,5]. The AUGMENT precompiler will generate the necessary calls to the routines in the interval package. Fortran programs which contain interval variables are compiled on MULTICS by use of the intfor command which will automatically invoke the AUGMENT precompiler for the user, see Appendix E. A description is given first of how to define the interval data type in Fortran and how to use it in a program. Next a detailed description of the I/O routines

for interval numbers will be given. A sample Fortran program illustrating the interval data type can be found in Appendix D.

3.1 Definition and Use of the Interval Data Type

A variable is declared in a Fortran program as having the interval data type through the use of an interval type declaration statement. The key word for the interval type declaration statement is "INTERVAL". For example, if the statement

```
INTERVAL A,B,C(10)
```

appeared in the Fortran program, then the scalar variables A and B would have the type interval and C would be an interval vector of 10 elements. The key word "INTERVAL" can appear in any context that the standard type key words (i.e. REAL, DOUBLE PRECISION, etc.) can appear. For example, if the statement

```
IMPLICIT INTERVAL (A-Z)
```

appeared in the Fortran program, then all variables beginning with the letters A-Z would default to type interval.

All of the standard arithmetic operators are defined for the interval data type. All implicit conversions from interval to the standard types and from the standard types to interval are defined except for conversion from logical to interval and interval to logical. The interval data type always takes precedence in an implicit conversion. All relational operators are defined between interval data types. All cases of exponentiation between the interval data type and the standard types are defined except for the standard types logical and complex. There are two new operators that act on interval operands. These operators are .INSCT. which finds the intersection of two intervals and .UNION. which finds the union of two intervals. For example, if A, B, and C are of type interval, then the statement

```
A = B .INSCT. C
```

finds the intersection of B and C and assigns the result to A. If the intersection is empty then an error message is displayed.

Most of the builtin functions of standard Fortran have been implemented for the type interval. The following are the builtin functions available for use with the interval data type.

ABS - Absolute value
ACOS- Arccosine

AINT - Integer
ALOG - Log base e
ALOG10 - Log base 10
ASIN - Arcsine
ATAN - Arctangent
ATAN2 - Arctangent of x/y
COS - Cosine
COSH - Hyperbolic cosine
DBLE - Converts to double precision
EXP - Exponential
FLOAT - Converts to real
IFIX - Converts to integer
SIN - Sine
SINH - Hyperbolic sine
SQRT - Square root
TAN - Tangent
TANH - Hyperbolic tangent

All of the above builtin functions take interval arguments and return an interval result except for the IFIX, FLOAT, and DBLE functions which return respectively type INTEGER, REAL, and DOUBLE PRECISION. The FLOAT and DBLE functions return the midpoint of the interval argument as either a real or double precision number. The IFIX function returns the integer portion of the midpoint of the interval argument. The AINT function computes an interval with integer endpoints (in floating point form) which contains the interval argument.

Several builtin functions were added for the type interval. These functions are listed and described below.

COMPOS: This function takes two real arguments and returns an interval with the first argument as the left endpoint and the second argument as the right endpoint.

DIST: This function takes two interval arguments and returns the distance between the intervals as a real number.

INF: This function takes one interval argument and returns its left endpoint as a real number.

INTBND: This function takes two arguments with the first argument being double precision and the second argument integer and returns an interval that contains the double precision number. The integer specifies the number of bits of accuracy of the double precision number.

INTSCT: This function takes two interval arguments and returns their intersection. Note that INTSCT can also be used as a binary operator as described earlier.

LENGTH: This function takes one interval argument and returns its length as a real number.

SUP: This function takes one interval argument and returns its right endpoint as a real number.

UNION: This function takes two interval arguments and returns their union. Note that UNION can also be used as a binary operator as described earlier.

If a constant is to be assigned to an interval variable, then in some cases the following type of assignment statement should be used.

```
A = "interval constant"
```

where an interval constant is defined in Section 3.2. This type of assignment statement should be used if 1) the interval being assigned is not degenerate or 2) a degenerate interval is being assigned, but there will be conversion error when converting from floating decimal to floating binary. An example of this type of assignment is contained in the sample interval Fortran program in Appendix D even though it was not necessary in that case to perform that type of assignment.

3.2 Interval Input/Output Routines

An interval constant consists of 1) a pair of floating point or fixed point numeric constants enclosed in parenthesis or square brackets and separated by a comma or 2) a single floating point or fixed point numeric constant that represents a degenerate interval. The form of the floating point or fixed point numeric constant is any number acceptable as a floating point decimal numeric constant in PL/I with a maximum of 59 decimal digits. Examples of interval constants are shown below:

```
(1,2)
[3,4]
1
( 1.0e15, 3.49 )
234.08e15
0.1
(-5,-4)
[-0.1,.6]
(-328.42, 2.3e-14)
( .1,.2 )
```

An interval number enclosed in parenthesis or square brackets may have any number of blanks before or after the parenthesis or square brackets and comma. The numbers representing the

endpoints may not have any embedded blanks. An interval number that represents a degenerate interval may not have any embedded blanks. Note that a decimal number that represents a degenerate interval may not be converted into a degenerate interval internally. This is because not all fractional decimal numbers have an exact representation in floating binary. For example, the decimal number 0.1 does not have a finite representation in floating binary. Therefore the endpoints of the resultant machine representable interval will not be equal because the left endpoint must be rounded downward and the right endpoint must be rounded upward when the decimal number is converted to floating binary.

Input Routines

The following three routines are used to read interval numbers. The numbers are input from Fortran file number 5 which is the standard Fortran input file. Any number of interval numbers can appear on each input line with 1 or more blanks separating the interval numbers. The interval numbers can be input from the terminal or from a segment through an appropriate operating system I/O attach statement.

Routine: intrdv

Purpose: Read interval numbers into any number of interval scalar variables.

Calling sequence: call intrdv (a,b,c,...,eof)

a,b,c,... (output) are interval scalar variables into which the interval numbers are to be read.

eof (output) is a logical variable that is true if end of file is encountered and is false if not.

Examples:

In the following examples assume a, b, c, and d are interval scalar variables and eof is a logical variable.

Example 1:

call intrdv (a,b,eof)

The next two interval numbers in the input stream will be read into the interval variables a and b.

Example 2:

call intrdv (a,b,c,d,eof)

The next four interval numbers in the input stream are read into the interval scalar variables a, b, c, and d.

In both of the above examples if end of file was detected, then the variable eof would be set to true, otherwise it is set to false.

Routine: intrdf

Purpose: Read interval numbers into an interval vector.

Calling sequence: call intrdf (x,i,j,eof)

x (output) is an interval vector.

i (input) is an integer variable or constant specifying the starting index of where in x the interval numbers will be placed.

j (input) is an integer variable or constant specifying the ending index of where in x the interval numbers will be placed.

eof (output) is a logical variable that is true if end of file is encountered and is false if not.

Examples:

In the following examples assume x is an interval vector that can contain a maximum of 10 interval numbers and eof is a logical variable.

Example 1:

call intrdf (x,1,10,eof)

The next ten interval numbers in the input stream will be read into the entire vector x.

Example 2:

call intrdf (x,2,7,eof)

The next 6 interval numbers in the input stream will be read into the interval locations x(2) to x(7).

In the both examples above, if end of file is encountered eof will be set to true, otherwise eof will be false.

Routine: intrdm

Purpose: Read interval numbers into an interval matrix in a row by row fashion.

Calling sequence: call intrdm (x,n,k,eof)

x (output) is an interval matrix.

n (input) is an integer variable or constant specifying the number of rows to be considered.

k (input) is an integer variable or constant specifying the number of columns to be considered.

eof (output) is a logical variable that is true if end of file is encountered and is false if not.

Examples:

In the following examples assume x is an interval matrix that can contain a maximum of 5 rows and 6 columns of interval numbers and eof is a logical variable.

Example 1:

call intrdm (x,5,6,eof)

The next 30 numbers in the input stream will be read into the entire matrix x row by row with 6 numbers per row and 5 total rows.

Example 2:

call intrdm (x,3,4,eof)

The next 12 interval numbers in the input stream will be read into the interval matrix x row by row with 4 numbers per row and 3 total rows.

In both examples above, if end of file is encountered, then eof is set to true, otherwise it is set to false.

Output_Routines

The following three routines are used to output interval numbers. The numbers are output to the PL/I standard output file, sysprint. The output can be directed to a segment either by an I/O attachment or through a file_output command.

Routine: intprv

Purpose: Output interval numbers from any number of interval scalar variables.

Calling sequence: call intprv (cc,nod,nob,width,a,b,c,....)

cc (input) is a single character representing the carriage control character. The carriage control characters are the same as in standard fortran and are listed below:

blank - single space
0 - double space
+ - suppress spacing
1 - skip to top of page

nod (input) is an integer variable or constant specifying the number of interval numbers to output per line.

nob (input) is an integer variable or constant specifying the number of blanks to insert between each interval number on each line.

width (input) is an integer variable or constant specifying the total width that each interval number will occupy in the output line. The interval number is output in the form [±.XX..XX±YY,±.XX..XX±YY] The number of significant digits output for each endpoint will be (width-13)/2.

a,b,c,... (input) are interval scalar variables to be output.

Examples:

In the following examples assume a, b, c, and d are interval variables.

Example 1:

call intprv (1h ,3,1,25,a,b,c)

The interval numbers in the interval variables a, b, and c will be output with single spacing. All three numbers will be on the same line. One blank will be between each interval number. Each interval number will occupy 25 columns providing 6 significant digits for each endpoint.

Example 2:

```
call intprv ("0",3,5,33,a,b,c,d)
```

The interval numbers in the interval variables a, b, c, and d will be printed with double spacing. a, b and c will be on one line with d on the next line. There will be five spaces between each interval number. Each interval number will occupy 33 columns providing 10 significant digits for each endpoint.

Routine: intpr

Purpose: Output interval numbers from an interval vector.

Calling sequence: call intpr (cc,nod,nob,width,x,i,j)

cc, nod, nob, and width are the same as for intprv.

x (input) is an interval vector.

i (input) is an integer variable specifying the starting index in the interval vector x from where output is to start.

j (input) is an integer variable specifying the ending index in the interval vector x where output is to stop.

Examples:

In the following examples assume x is an interval vector that can contain a maximum of 10 interval numbers.

Example 1:

```
call intpr (" ",2,10,35,x,1,10)
```

The interval numbers in the entire interval vector x will be output with single spacing. There will be two interval numbers per line. Ten blanks will be between each interval number. Each interval number will occupy 35 columns providing 11 significant digits for each endpoint.

Example 2:

```
call intpr (1h0,5,1,25,x,3,9)
```

The 7 interval numbers in interval locations x(3) to x(9) will be printed with double spacing. There will be five interval numbers per line. One blank will be between each interval number. Each interval number will occupy 25 columns providing 6 significant

digits for each endpoint.

Routine: intprm

Purpose: Print interval numbers from an interval matrix.

Calling sequence: call intprm (cc,nod,nob,width,x,n,k)

cc, nod, nob, and width are the same as for intprv.

x (input) is an interval matrix.

n (input) is an integer variable or constant specifying the number of rows of the interval matrix to output.

k (input) is an integer variable or constant specifying the number of columns of the interval matrix to output.

Examples:

In the following examples assume x is an interval matrix that can contain a maximum of 7 rows and 5 columns of interval numbers.

Example 1:

call intprm (" ",5,1,25,x,7,5)

The interval numbers in the entire interval matrix x will be printed with single spacing. There will be five interval numbers per line. One blank will be between each interval number. Each interval number will occupy 25 columns providing 6 significant digits for each endpoint.

Example 2:

call intprm ("0",4,5,27,x,6,4)

The interval numbers in the first 6 rows and 4 columns of the interval matrix x will be printed with double spacing. There will be four interval numbers per line. Five blanks will be between each interval number. Each interval number will occupy 27 columns providing 7 significant digits for each endpoint.

References

[1] Ladner, T. D. and Yohe, J. M., "An interval arithmetic package for the UNIVAC 1108," The University of

Wisconsin, Mathematics Research Center, Technical Summary Report No. 1055, May, 1970.

[2] Yohe, J. M., "Best possible floating point arithmetic," The University of Wisconsin, Mathematics Research Center, Technical Summary Report No. 1054, March, 1970.

[3] Binstock, W., Hawkes, J. and Hsu, N., "An interval input/output package for the UNIVAC 1108," The University of Wisconsin, Mathematics Research Center, Technical Summary Report No. 1212, September, 1973.

[4] Crary, F. D., "The AUGMENT precompiler, I. User information," The University of Wisconsin, Mathematics Research Center, Technical Summary Report No. 1469, December, 1974.

[5] Crary, F. D., "The AUGMENT precompiler, II. Technical documentation," The University of Wisconsin, Mathematics Research Center, Technical Summary Report No. 1470, October, 1975.

[6] Moore, R. E, Interval Analysis, Prentice-Hall Inc., Englewood Cliffs, N. J., 1966.

[7] Abramovitz, M. and Stegun, I. A., (ed.), Handbook of Mathematical Functions, National Bureau of Standard Applied Mathematics Series, June, 1964.

[8] Reuter, E. K. and Podlaska-Lando, S., "Source Listing for the MULTICS Interval Arithmetic Package," Computer Science Department Report No. 76-7-3, University of Southwestern Louisiana, Lafayette, Louisiana, August, 1976.

Appendix A - Mathematical Basis for Interval Arithmetic

The details of the mathematical basis for interval arithmetic are developed in Moore [6]. The set of interval numbers is the set of all closed intervals on the real number line. An interval may be represented by an ordered pair of real numbers [a,b] where $a \leq b$. If $a = b$, then the interval is said to be degenerate.

The operations of addition, subtraction, multiplication, and division between two intervals (except for the division of one interval by an interval containing zero) are defined as follows where \$ is one of the above operations:

$$[a,b] \$ [c,d] = \{x \$ y : x \in [a,b] \text{ and } y \in [c,d]\}$$

Each of the operations of addition, subtraction, multiplication, and division may be defined as follows:

$$[a,b] + [c,d] = [a+c, b+d]$$

$$[a,b] - [c,d] = [a-d, b-c]$$

$$[a,b] * [c,d] = [\min\{ac,ad,bc,bd\}, \max\{ac,ad,bc,bd\}]$$

$$[a,b] / [c,d] = [\min\{a/c, a/d, b/c, b/d\}, \max\{a/c, a/d, b/c, b/d\}] \quad \text{if } 0 \notin [c,d]$$

In the cases of multiplication and division, by examining the signs of the endpoints of the intervals being multiplied or divided; a determination in advance can be made of which products or quotients will be the maximum and the minimum.

The following real single valued functions of intervals may be useful:

The midpoint of an interval, mid ([a,b]), is defined to be the real number (a+b)/2.

The length of an interval, length ([a,b]), is defined to be the real number b-a.

The supremum of an interval, sup ([a,b]), is the real number a.

The infimum of an interval, inf ([a,b]), is the real number b.

The distance from interval [a,b] to interval [c,d], dis ([a,b],[c,d]), is defined to be the real number $\max\{|c-a|, |d-b|\}$.

The following interval single valued functions of intervals may also be useful:

The union of intervals [a,b] and [c,d], union ([a,b],[c,d]), is defined to be the smallest interval containing both [a,b] and [c,d] and is given by [min(a,c),max(b,d)]. The intersection of intervals [a,b] and [c,d], intsct ([a,b],[c,d]), is defined to be the largest interval contained in each of [a,b] and [c,d] or is empty if [a,b] and [c,d] are disjoint intervals and is given by [max(a,c),min(c,d)].

The relational operations may be defined on intervals as follows:

$$[a,b] = [c,d] \text{ if and only if } a = b = c = d$$

The above definition means that two intervals are equal if and only if they both are degenerate and represent the same real number. This definition is employed instead of the more general definition of testing for a = c and b = d. The reason the more general definition is not used is because we will regard intervals as bounds on an exact but unknown real number. If two intervals were not degenerate and if both intervals had the same endpoints, then the intervals may not represent the same exact real number. The only way for the two intervals to represent the same exact real number is for both intervals to be degenerate with their endpoints equal to the real number. We also say that

$$[a,b] \neq [c,d] \text{ if and only if } [a,b] \text{ intersection } [c,d] = \emptyset$$

This definition means that two intervals are not equal if and only if they are disjoint intervals and cannot represent the same exact real number.

$$[a,b] \leq [c,d] \text{ if and only if } b \leq c$$

The above definition means that two intervals are ordered by the $\leq$ relational operator if and only if $\forall x \in [a,b]$ and $\forall y \in [c,d]$, $x \leq y$.

$$[a,b] > [c,d] \text{ if and only if } a > d$$

The above definition means that two intervals are ordered by the $>$ relational operator if and only if $\forall x \in [a,b]$ and $\forall y \in [c,d]$, $x > y$.

Interval valued functions of interval variables are defined in terms of real valued functions of real variables. If f is a real valued function of a real variable, then f may be extended to an interval valued function, F, of an interval variable by defining

$$F([a,b]) = \{f(x) : x \in [a,b]\}$$

If f is defined and continuous on [a,b], then F([a,b]) will be an interval. If intervals are to represented as pairs of real numbers, then the above definition is not operational. Some

means is needed for deriving the endpoints of the image of [a,b] under the function F. The endpoints of the image interval will be the image under f of points of [a,b].

For functions, f, that are monotonic on the interval [a,b], the endpoints of the image of [a,b] under F can be expressed as the result of the function f evaluated at the endpoints of [a,b]. If f is monotonic increasing on [a,b], then F([a,b]) = [f(a), f(b)]. If f is monotonic decreasing on [a,b], then F([a,b]) = [f(b), f(a)]. If f is not monotonic over [a,b], then [a,b] can be divided into disjoint subintervals; X(i), i = 1,2,3,...n; where U X(i) = [a,b] and f is monotonic on each X(i). In this case F([a,b]) = U f(X(i)).

Appendix B - Description of Interval Faults and List of Error Messages

The following table lists the possible fault conditions that can arise during an interval operation along with the value of the fault flag and the default action code that specifies the action taken by intrap after it is called. The action code is explained after the table.

Fault Flag	Fault Condition		Default Response
	Left Endpoint	Right Endpoint	
0	no faults	no faults	-
1	no faults	overflow	3
2	no faults	infinity	2
3	no faults	underflow	0
4	overflow	no faults	3
5	overflow	overflow	3
6	overflow	infinity	2
7	overflow	underflow	3
8	infinity	no faults	2
9	infinity	overflow	2
10	infinity	infinity	2
11	infinity	underflow	2
12	underflow	no faults	0
13	underflow	overflow	3
14	underflow	infinity	2
15	underflow	underflow	0
16	division by zero		2
17	zero to the zero power		1
18	square root of a negative number		2
19	log of a non-positive number		2
20	underflow during interval-to-real		0
21	overflow during interval-to-real		2
22	intersection of disjoint intervals		2
23	argument out of range		2
24	underflow during interval-to-double		2
25	underflow		2
26	overflow		2

The action codes are as follows:

0 - Exit
1 - Print error message and arguments
2 - Print error message, arguments and trace stack
3 - Print error message, arguments, trace stack and stop

The arguments that are displayed are the arguments of the calling program. Three arguments are always passed to intrap. If the calling program had only two arguments, then the first two arguments passed to intrap have the same value as the first argument of the calling routine. If the calling routine has only one argument, then all three arguments passed to intrap have the same value as that argument.

intrap_Modification

Under certain circumstances the user may wish to change the default action taken by intrap when a fault occurs. The user may also wish to change the value assigned as the result of an operation in order to be more mathematically consistent with the problem to be solved. The user can modify the action taken by intrap by including the following statements in the user's Fortran program.

```
common /intflt/ ifault,routin,type(3),itgarg(3),rarg(3),darg(3),
                itvagr(2,3),montor(32)
integer type
character*6 routin
real itvarg
double precision darg
```

"ifault" will contain the fault flag after each operation.

"routin" will contain a character string which is the name of the last routine to call intrap.

"type" will contain the types of the last three arguments passed to intrap. If type(i) is zero, then that particular argument was not present in the call to intrap. The type codes are as follows:

1 - integer
2 - real
3 - double precision
4 - interval

"itgarg", "rarg", "darg", and "itvarg" will contain the arguments passed to intrap. They contain respectively either the integer, real, double precision, or interval arguments. For example, if type(1) = 3, type(2) = 1, and type(3) = 4 then darg(1) will

contain the double precision argument, itgarg(2) will contain the integer argument, and itvarg(1,3) will contain the interval argument.

"montor" contains the action codes for each type of fault listed in the table. If the user wants to change the action for a particular fault, then the user changes the location in the montor array that corresponds to the particular fault. For example, if the user wishes to change the action for a divide by zero fault to a fatal error, then the following statement is included in the user's program.

```
montor(16) = 3
```

In this case when a zero divide occurs the program will stop.

List_of_Error_Messages

The following is a list of the error messages produced by intrap.

BOUNDS FAULT DURING "routine name" LEFT ENDPOINT-- "fault" RIGHT ENDPOINT-- "fault"

DIVISION BY ZERO DURING "routine name"

ZERO TO THE ZERO POWER DURING "routine name"

SQUARE ROOT OF A NEGATIVE NUMBER DURING "routine name"

LOG OF A NON-POSITIVE NUMBER DURING "routine name"

UNDERFLOW DURING CONVERSION FROM INTERVAL TO REAL IN "routine name"

OVERFLOW DURING CONVERSION FROM INTERVAL TO INTEGER IN "routine name"

INTERSECTION OF DISJOINT INTERVALS DURING "routine name"

ARGUMENT OUT OF RANGE IN "routine name"

UNDERFLOW DURING CONVERSION FROM INTERVAL TO DOUBLE PRECISION IN "routine name"

UNDERFLOW IN "rouitne name"

OVERFLOW IN "routine name"

UNKNOWN ERROR DURING "routine name"

Appendix C - Summary of Interval Subroutine Modification

The following is a list of the routines written for the interval package divided into the categories with the codes [MRC], [MRC/M], and [MUL].

[MRC] - Original form from the Mathematics Research Center with no modifications.

arith2	expon1
expon2	convrt
intc87	relatn
supinf	unints
intbnd	funct3
comput	

[MRC/M] - Original form with modifications.

arith1	bpaxp4
intc84	intc85
intc86	funct1
length	funct2
funct4	funct5

[MUL] - Written specifically for the MULTICS System.

bpaadd	bpasub	bpamul
bpadiv	brounding	unpack
normalize	shift_right	s_mgn_add
aidint	pack	dist
intrdv	intrdf	intrdm
intprv	intpr	intprm
convert_to_binary	convert_to_decimal	convert_fb_dec
get_next_int_number	round_dec	get_char
set_input_pointer	intrap	bpac68
bpac98	intas	set_common
finish		

Appendix D - Sample Interval Fortran Program

The following Fortran program illustrates the use of the interval data type. The program solves a set of linear equations using Gaussian elimination with partial pivoting. The program is just illustrative of the interval data type and may not be the best method of solving a set of linear equations using interval arithmetic. A listing of the interval program is given followed by a listing of the translated program produced by AUGMENT with the calls to the interval package routines. Following that is a sample of the output produced by the program. In one case the error trapping capability of the interval package is illustrated. In this case it was set up so that if an error occurred the program continued.

```
      INTERVAL A(10,11),X(10),BIG,TERM,PIVOT,CONST,Y,TEMP
      LOGICAL EOF
      INTEGER YES
      DATA YES /3Hyes/
  910 WRITE(6,710)
  710 FORMAT(/," ENTER NUMBER OF EQUATIONS")
      READ(5,10)NN
   10 FORMAT(V)
      M=NN
      N=M+1
      WRITE(6,750)
  750 FORMAT(/," ENTER MATRIX")
      CALL INTRDM(A,M,N,EOF)
      LAST=M-1
C
C     START OVERALL LOOP FOR M-1 PIVOTS
C
      DO 200 I=1,LAST
C
C     FIND LARGEST REMAINING TERM IN I-TH COLUMN FOR PIVOT
C
      BIG="0"
      DO 50 K=I,M
      TERM=ABS(A(K,I))
      IF(TERM.LE.BIG) GO TO 50
      BIG=TERM
      L=K
   50 CONTINUE
C
C     CHECK WHETHER A NON-ZERO TERM HAS BEEN FOUND
C
      IF(BIG.EQ."0") STOP
C
C     L-TH ROW HAS THE BIGGEST TERM -- IS I=L
C
      IF(I.EQ.L) GO TO 120
```

```
C
C       I IS NOT EQUAL TO L, SWITCH ROWS I AND L
C
        DO 100 J=1,N
        TEMP=A(I,J)
        A(I,J)=A(L,J)
100     A(L,J)=TEMP
C
C       NOW START PIVOTAL REDUCTION
C
120     PIVOT=A(I,I)
        NEXTR=I+1
C
C       FOR EACH OF THE ROWS AFTER THE I-TH
C
        DO 200 J=NEXTR,M
C
C       CONST IS MULTIPLYING CONSTANT FOR THE J-TH ROW
C
        CONST=A(J,I)/PIVOT
C
C       NOW REDUCE EACH TERM OF THE J-TH ROW
C
        DO 200 K=I,N
200     A(J,K)=A(J,K)-CONST*A(I,K)
C
C       END OF PIVOTAL REDUCTION - PRINT REDUCED MATRIX
C
        WRITE(6,501)
501     FORMAT(/," THE REDUCED MATRIX IS AS FOLLOWS:",/)
        CALL INTPRM(1H ,3,1,25,A,M,N)
C
C       PERFORM BACK SUBSTITUTION
C
        DO 500 I=1,M
C
```

```
C       IREV IS THE BACKWARD INDEX, GOING FROM M BACK TO 1
C
        IREV=M+1-I
C
C       GET Y IN PREPARATION
C
        Y=A(IREV,N)
        IF(IREV.EQ.M) GO TO 500
C
C       NOT WORKING ON LAST ROW, I IS 2 OR GREATER
C
        DO 450 J=2,I
C
C       WORK BACKWARD FOR X(N), X(N-1) ..., SUBSTITUTING PREVIOUSLY FOUND
C       VALUES
C
        K=N+1-J
450     Y=Y-A(IREV,K)*X(K)
C
C       FINALLY COMPUTE X
C
500     X(IREV)=Y/A(IREV,IREV)
C
C       PRINT VALUES OF X
C
        WRITE(6,502)
502     FORMAT(/," THE SOLUTION IS AS FOLLOWS:",/)
        CALL INTPR(1H ,1,0,25,X,1,M)
        WRITE(6,950)
950     FORMAT(/," DO YOU WANT TO CONTINUE?")
        READ(5,975)IRESP
975     FORMAT(A3)
        IF(IRESP.EQ.YES) GO TO 910
        ENDFILE 5
        ENDFILE 6
        STOP
```

```
END
```

```
C                ===== PROCESSED BY AUGMENT VERSION ,4I =====
C                ----- TEMPORARY STORAGE LOCATIONS -----
C                      INTERVAL
      REAL INTTMP(2,1)
C                ----- LOCAL VARIABLES -----
      LOGICAL EOF
      INTEGER I, IRESP, IREV, J, K, L, LAST, M, N, NEXTR, NN, YES
C                      INTERVAL
      REAL A(2,10,11), BIG(2), CONST(2), PIVOT(2), TEMP(2), TERM(2), X(2
     *     ,10), Y(2)
C                ----- SUPPORTING PACKAGE FUNCTIONS -----
      LOGICAL INTEQ, INTLE
C                ===== TRANSLATED PROGRAM =====
C ===== DATA STATEMENTS ARE NOT PROCESSED BY AUGMENT =====
      DATA YES /3Hyes/
910   WRITE(6,710)
710   FORMAT(/," ENTER NUMBER OF EQUATIONS")
      READ(5,10)NN
10    FORMAT(V)
      M=NN
      N=M+1
      WRITE(6,750)
750   FORMAT(/," ENTER MATRIX")
      CALL INTRDM(A,M,N,EOF)
      LAST=M-1
C
C     START OVERALL LOOP FOR M-1 PIVOTS
C
      DO 30000 I=1,LAST
C
C     FIND LARGEST REMAINING TERM IN I-TH COLUMN FOR PIVOT
C
      CALL INTAS ("0",BIG)
      DO 50    K=I,M
      CALL INTABS (A(1,K,I),TERM)
      IF (INTLE (TERM,BIG))  GO TO 50
```

```
      CALL INTSTR (TERM,BIG)
      L=K
50    CONTINUE
C
C     CHECK WHETHER A NON-ZERO TERM HAS BEEN FOUND
C
C ===== MIXED MODE OPERANDS ACCEPTED =====
      CALL INTAS ("0",INTTMP(1,1))
      IF (INTEQ (BIG,INTTMP(1,1)))  STOP
C
C     L-TH ROW HAS THE BIGGEST TERM -- IS I=L
C
      IF(I.EQ.L) GO TO 120
C
C     I IS NOT EQUAL TO L, SWITCH ROWS I AND L
C
      DO 100   J=1,N
      CALL INTSTR (A(1,I,J),TEMP)
      CALL INTSTR (A(1,L,J),A(1,I,J))
100   CALL INTSTR (TEMP,A(1,L,J))
C
C     NOW START PIVOTAL REDUCTION
C
120   CALL INTSTR (A(1,I,I),PIVOT)
      NEXTR=I+1
C
C     FOR EACH OF THE ROWS AFTER THE I-TH
C
      DO 30000 J=NEXTR,M
C
C     CONST IS MULTIPLYING CONSTANT FOR THE J-TH ROW
C
      CALL INTDIV (A(1,J,I),PIVOT,CONST)
C
C     NOW REDUCE EACH TERM OF THE J-TH ROW
C
```

```
      DO 30000 K=I,N
200   CALL INTMUL (CONST,A(1,I,K),INTTMP(1,1))
      CALL INTSUB (A(1,J,K),INTTMP(1,1),A(1,J,K))
30000 CONTINUE
C
C     END OF PIVOTAL REDUCTION - PRINT REDUCED MATRIX
C
      WRITE(6,501)
501   FORMAT(/," THE REDUCED MATRIX IS AS FOLLOWS:",/)
      CALL INTPRM(1H ,3,1,25,A,M,N)
C
C     PERFORM BACK SUBSTITUTION
C
      DO 500   I=1,M
C
C     IREV IS THE BACKWARD INDEX, GOING FROM M BACK TO 1
C
      IREV=M+1-I
C
C     GET Y IN PREPARATION
C
      CALL INTSTR (A(1,IREV,N),Y)
      IF(IREV.EQ.M) GO TO 500
C
C     NOT WORKING ON LAST ROW, I IS 2 OR GREATER
C
      DO 30001 J=2,I
C
C     WORK BACKWARD FOR X(N), X(N-1) ..., SUBSTITUTING PREVIOUSLY FOUND
C     VALUES
C
      K=N+1-J
450   CALL INTMUL (A(1,IREV,K),X(1,K),INTTMP(1,1))
      CALL INTSUB (Y,INTTMP(1,1),Y)
30001 CONTINUE
C
```

```
C       FINALLY COMPUTE X
C
500     CALL INTDIV (Y,A(1,IREV,IREV),X(1,IREV))
C
C       PRINT VALUES OF X
C
        WRITE(6,502)
502     FORMAT(/," THE SOLUTION IS AS FOLLOWS:",/)
        CALL INTPR(1H ,1,0,25,X,1,M)
        WRITE(6,950)
950     FORMAT(/," DO YOU WANT TO CONTINUE?")
        READ(5,975)IRESP
975     FORMAT(A3)
        IF(IRESP.EQ.YES) GO TO 910
        ENDFILE 5
        ENDFILE 6
        STOP
        END
```

```
ENTER NUMBER OF EQUATIONS
2

ENTER MATRIX
5 3 6
1 8 4

THE REDUCED MATRIX IS AS FOLLOWS:

 [ .500000+01, .500000+01] [ .300000+01, .300000+01] [ .600000+01, .600000
 [-.149012-07, .745059-08] [ .739999+01, .740001+01] [ .279999+01, .280001

THE SOLUTION IS AS FOLLOWS:

[ .972972+00, .972973+00]
[ .378378+00, .378379+00]

DO YOU WANT TO CONTINUE?
yes

ENTER NUMBER OF EQUATIONS
3

ENTER MATRIX
5 3 6 8
2 8 3 1
9 4 6 2

THE REDUCED MATRIX IS AS FOLLOWS:

 [ .900000+01, .900000+01] [ .400000+01, .400000+01] [ .600000+01, .600000
 [ .200000+01, .200000+01]
 [-.298024-07, .149012-07] [ .711111+01, .711112+01] [ .166666+01, .166667
 [ .555555+00, .555556+00]
 [-.596047-07, .596047-07] [-.447035-07, .447035-07] [ .248437+01, .248438
 [ .682812+01, .682813+01]
```

```
THE SOLUTION IS AS FOLLOWS:

[-.135850+01,-.135849+01]
[-.566038+00,-.566037+00]
[ .274842+01, .274843+01]

DO YOU WANT TO CONTINUE?
yes

ENTER NUMBER OF EQUATIONS
2

ENTER MATRIX
[1,2] [5,6] [8,9]
[12,13] [3,4] [15,16]

THE REDUCED MATRIX IS AS FOLLOWS:

 [ .120000+02, .130000+02] [ .300000+01, .400000+01] [ .150000+02, .1600
 [-.116667+01, .107693+01] [ .433333+01, .576924+01] [ .533333+01, .7846

THE SOLUTION IS AS FOLLOWS:

[ .596722+00, .110223+01]
[ .924444+00, .181066+01]

DO YOU WANT TO CONTINUE?
yes

ENTER NUMBER OF EQUATIONS
2

ENTER MATRIX
1 1 1
2 2 2
```

```
THE REDUCED MATRIX IS AS FOLLOWS:

 [ .200000+01, .200000+01] [ .200000+01, .200000+01] [ .200000+01, .20000
 [ .000000+00, .000000+00] [ .000000+00, .000000+00] [ .000000+00, .00000

**********************************************************************
DIVISION BY ZERO DURING intdiv
ARGUMENT 1 = [ .00000000000000000000000000+00, .00000000000000000000000000
ARGUMENT 2 = [ .00000000000000000000000000+00, .00000000000000000000000000
RESULT =     [-.17014118219281863150345791+39, .17014118219281863150345791
**********************************************************************

**********************************************************************
BOUNDS FAULT DURING intmul LEFT ENDPOINT--INFINITY RIGHT ENDPOINT--INFINI
ARGUMENT 1 = [ .20000000000000000000000000+01, .20000000000000000000000000
ARGUMENT 2 = [-.17014118219281863150345791+39, .17014118219281863150345791
RESULT =     [-.17014118219281863150345791+39, .17014118219281863150345791
**********************************************************************

**********************************************************************
BOUNDS FAULT DURING intsub LEFT ENDPOINT--NO FAULTS RIGHT ENDPOINT--INFIN
ARGUMENT 1 = [ .20000000000000000000000000+01, .20000000000000000000000000
ARGUMENT 2 = [-.17014118219281863150345791+39, .17014118219281863150345791
RESULT =     [-.17014118219281863150345791+39, .17014118219281863150345791
**********************************************************************

THE SOLUTION IS AS FOLLOWS:

[-.850706+38, .850706+38]
[-.170142+39, .170142+39]

DO YOU WANT TO CONTINUE?
no
STOP
```

Appendix E - Use of the intfor Command on MULTICS

Function: translates interval fortran programs into standard Fortran and compiles the translated segment if requested.

Syntax: intfor path -control_args-

Arguments: path is the pathname of an interval Fortran source segment; a suffix of ".interval" is assumed and need not be given.

Control arguments: -no_translated_source, -nts does not create the translated Fortran segment in the current working directory; default is to create a translated source segment with the suffix ".fortran". Any error messages produced during translation will be in this segment (see Notes below).

-convert_real_to_interval, -cri all variables of type real in the source will be considered to have the type interval.

-no_compile, -nc the translated source will not be compiled; default is to compile.

-force_compile, -fc there will be an attempt to compile the translated source even if there are errors during the translation.

-augment_list, -als a segment is produced by the AUGMENT precompiler (see Notes below) that consists of a listing of the input source segment passed to AUGMENT. Any error messages produced during translation will also be in this segment. The segment will have the suffix "agm.list"

The rest of the control arguments are any arguments acceptable to the Fortran compiler. These arguments will be passed to the Fortran compiler if a compilation is to be performed.

Notes: The intfor command uses the AUGMENT precompiler to produce the translated source. AUGMENT will display how many errors there were in the processing or translation phase. The error messages will be in the translated source segment next to the statement that caused the error. Therefore if a program is being translated for the first time, a translated source segment should be created in case there are errors. The error messages can be located in the translated source segment by searching for the character string "*****" which is attached to each error message.

Currently the input segment must be in the standard Fortran 80 column format using all uppercase letters.

In order to run an interval program the user must have the search rules set to search the directory >udd>beta>r&d>a.r&d>int_routines. This setting of the search rules can be done by a "ssr >udd>beta>r&d>a.r&d>isr" before running the interval program.